UNDERSTANDING DATA STRUCTURES

Rob Gerritsen

Carnegie-Mellon University

Prepared for:

Air Force Office of Scientific Research
Advanced Research Projects Agency

February 1975

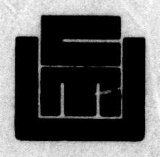
UNDERSTANDING DATA STRUCTURES

by

Rob Gerritsen

February 1975

DEPARTMENT of COMPUTER SCIENCE



Carnegie-Mellon University

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER AFOSR - TR - 75 - 0545	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER AD-A008 937
4. TITLE (and Subtitle) UNDERSTANDING DATA STRUCTURES		5. TYPE OF REPORT & PERIOD COVERED Interim
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Rob Gerritsen		8. CONTRACT OR GRANT NUMBER(s) F44620-73-C-0074
9. PERFORMING ORGANIZATION NAME AND ADDRESS Carnegie-Mellon University Department of Computer Science Pittsburgh, PA 15213		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 61101D A02466
11. CONTROLLING OFFICE NAME AND ADDRESS Defense Advanced Research Projects Agency 1400 Wilson Blvd Arlington, VA 22209		12. REPORT DATE February 1975
		13. NUMBER OF PAGES 230
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) Air Force Office of Scientific Research/NM 1400 Wilson Blvd Arlington, VA 22209		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Data management programmers are finding their jobs are getting tougher because of the gradual replacement of sequential data bases by network data bases. In addition, there is a new job called "Data Administrator" for handling the data structure problems associated with network data bases. The goal of this thesis is reduction of these data management tasks by developing and applying a practical theory of data structure. To insure the practical flavor of this research, the Data Base Task Group (DBTG) report has been selected as the specification of the data management system in

(CONTINUED)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

20. abstract (CONTINUED)

which the applications function.

An implemented system that automates both information retrieval and data base design demonstrates the application of the theory. To do this, the theory is captured in a Frame, a set of formal rules in the logic of programs. An Automatic Programming Generator(APG) compiles this Frame to an operating program. Two frames, resulting in two programs, are discussed. One of these programs generates information retrieval procedure. The other program designs a data base structure.

Both of these programs accept relational queries (expressed in the HI-IQ language) as input. These programs can be viewed as translators from relational descriptions to access path descriptions.

Programming and data structure design are cerebral tasks. Deriving the Frames whereby the programs understand data structure so that these tasks can be replicated was difficult. These difficulties are discussed for the benefit of others who want to apply Artificial Intelligence.

This research is justified in practical terms: A comparison of manual and automatic programming costs shows a potential cost reduction of up to 98% for automatic programming.

UNDERSTANDING DATA STRUCTURES

by

Rob Gerritsen

February 1975

DISSERTATION

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Industrial Administration (Systems).

Graduate School of Industrial Administration

Carnegie-Mellon University

Pittsburgh, Pennsylvania

This research was supported in part by the Advanced Research Projects Agency of the Office of the Secretary of Defense under contract F44620-73-C-0074. A William Larimer Mellon Fellowship provided personal funding.

TO MY PARENTS

Alexander N. Gerritsen
Jacqueline K. Gerritsen

TABLE OF CONTENTS

ACKNOWLEDGEMENT 3

ABSTRACT 4

LIST OF ILLUSTRATIONS 5

1 THEORY AND SUPPORT 7
- 1.1 Introduction 7
- 1.2 Relevant Literature 11
- 1.3 DBTG Data Bases 17
- 1.4 Matrices, Hierarchies, Relationships and Networks 22

2 DATA MANIPULATION LANGUAGE PROGRAMMER 27
- 2.1 Introduction 27
- 2.2 The APG System 35
- 2.3 Program Generation vs a Generalized Interpreter 42
- 2.4 HI-IQ, the Query Language 45
- 2.5 Request Handler Assertions 57
- 2.6 A BNF Description of the Generated Procedure 63
- 2.7 Assertions 77
- 2.8 A Frame for the Semantics of Data Structures 89
- 2.9 Efficiency Considerations 104
- 2.10 Algol to COBOL Conversion 109
- 2.11 Examples of Procedure Generation 115

3 DESIGN OF DATA STRUCTURES 134
- 3.1 Introduction 134
- 3.2 Use of the Programmer 137
- 3.3 Automatic Data Structure Design 140
- 3.4 Defining Item Names 143
- 3.5 Generating Assertions 144
- 3.6 Designing Record Relationships 150
- 3.7 A Frame for Record Relationship Design 155
- 3.8 Designing Record Contents 164
- 3.9 Example 169
- 3.10 Alternative Implementation 180

4 EXTENSIONS 182
- 4.1 Extensions to the DMLP 182
- 4.2 Extensions to the Data Structure Designer 190
- 4.3 Automation of Data Base Update 192
- 4.4 Data Base Restructuring 194

ACKNOWLEDGEMENT

I thank Professor Jack R. Buchanan, my thesis supervisor and chairman of the committee; his suggestions and direction were essential throughout. I also thank Professor Charles H. Kriebel and Professor Herbert A. Simon, members of my thesis committee, for their excellent teaching and advice. Finally, I am deeply indebted to my wife, Joyce, who not only provided encouragement but also did most of the typing and proofreading.

Table of Contents

5 COST EFFECTIVENESS OF AUTOMATIC PROGRAMMING 197
5.1 Measures of Automatic Program Generation 197
5.2 Cost Factors in Programming 200
5.3 Comparison of Dollar Costs 202
5.4 Execution Costs 204
5.5 Cost of Data Structure Design 205

6 ON THE APPLICATION OF ARTIFICIAL INTELLIGENCE 207
6.1 A Data Management Application 208
6.2 Acquisition and Representation of Knowledge 213

7 THE RELATIONAL AND NETWORK MODELS OF DATA BASES 217
7.1 Introduction 217
7.2 Levels of Data Structure Description 217
7.3 Translation Approach 219
7.4 Advantages of Translation 220
7.5 Two Implemented Translators 221

BIBLIOGRAPHY 222

ACKNOWLEDGEMENT

I thank Professor Jack R. Buchanan, my thesis supervisor and chairman of the committee; his suggestions and direction were essential throughout. I also thank Professor Charles H. Kriebel and Professor Herbert A. Simon, members of my thesis committee, for their excellent teaching and advice. Finally, I am deeply indebted to my wife, Joyce, who not only provided encouragement but also did most of the typing and proofreading.

ABSTRACT

Data management programmers are finding their jobs are getting tougher because of the gradual replacement of sequential data bases by network data bases. In addition, there is a new job called "Data Administrator" for handling the data structure problems associated with network data bases. The goal of this thesis is reduction of these data management tasks by developing and applying a practical theory of data structure. To insure the practical flavor of this research, the Data Base Task Group (DBTG) report has been selected as the specification of the data management system in which the applications function.

An implemented system that automates both information retrieval and data base design demonstrates the application of the theory. To do this, the theory is captured in a Frame, a set of formal rules in the logic of programs. An Automatic Programming Generator (APG) compiles this Frame to an operating program. Two frames, resulting in two programs, are discussed. One of these programs generates information retrieval procedure. The other program designs a data base structure.

Both of these programs accept relational queries (expressed in the Rt-IQ language) as input. These programs can be viewed as translators from relational descriptions to access path descriptions.

Programming and data structure design are cerebral tasks. Deriving the Frames whereby the programs understand data structure so that these tasks can be replicated was difficult. These difficulties are discussed for the benefit of others who want to apply Artificial Intelligence.

This research is justified in practical terms: A comparison of manual and automatic programming costs shows a potential cost reduction of up to 98% for automatic programming.

LIST OF ILLUSTRATIONS

1-1	Applications of data structure theory.	8
1-2	Simple DOCTOR-PATIENT data base.	19
1-3	Better DOCTOR-PATIENT data base.	20
1-4	Simple tabular reports.	23
1-5	Hierarchical report.	24
1-6	Hierarchical report (inverted from Figure 1-5).	26
2-1	Simple query with generated procedure.	29
2-2	System flows.	32
2-3	Average cost per usage for generated and interpreted programs in a production environment.	42
2-4	BNF for the HI-IQ language.	48
2-5	Two examples of retrieval conditions.	51
2-6	Query (P1) for the report of Figure 2-7.	53
2-7	A hierarchical-matrix report.	54
2-8	Templates for Request Handler assertions.	58
2-9	Possible values of the TYPE parameter in LINKS.	59
2-10	Assertions describing the query of Figure 2-6.	62
2-11	A BNF description of the generated procedure.	64
2-12	Code generated for the condition of Figure 2-5.	72
2-13	Assertions used to describe the data base.	78
2-14	Assertions which indicate the results of single program statements.	79
2-15	Other assertions.	81
2-16	Assertions used to describe program blocks.	83
2-17	Assertions evaluated by LISP.	85
2-18	Standard LISP and Micro-Planner predicates used in the rules.	87
2-19	Index to Figures 2-8 and 2-13 through 2-18.	88
2-20	Operator (type S1) rules.	91
2-21	The iteration rule.	94
2-22	Type S3 and S4 rules.	95
2-23	Rule and production correspondences.	99
2-24	Area search vs a SYSTEM owned set search.	105
2-25	Program P1 (Figure 2-6) in Algol.	113
2-26	Program P1 in COBOL.	114
2-27	Index to the examples.	116
2-28	A community medical data base structure.	117
2-29	A sales data base structure.	118
2-30	Query P2.	121
2-31	Program P2.	122
2-32	Query P3.	124
2-33	Program P3.	125
2-34	Query P4.	127

List of Illustrations

2-35	Program P4.	128
2-36	Query P5.	130
2-37	Program P5.	131
2-38	Query P6.	132
2-39	Program P6.	133
3-1	A simple user-programmer-DBA system.	138
3-2	Data base structure designer; general program flow.	142
3-3	Assertions and their meanings.	145
3-4	Transformation of a confluent hierarchy within the request context.	148
3-5	Construction of record relationships.	151
3-6	Rules for generating record relationships.	156
3-7	Assertions and their interpretations.	159
3-8	Assertions before and after application of the first three rules.	160
3-9	Some interesting structure transformations.	162
3-10	Determination of record content - general flow.	165
3-11	Rules for establishing record contents.	166
3-12	Query number one.	171
3-13	Query number two.	172
3-14	Query number three.	173
3-15	Query number four.	174
3-16	Query number five.	175
3-17	Query number six.	176
3-18	Assertions generated from queries 1-6	177
3-19	Generated data base structure.	178
3-20	Data structure diagram for Figure 3-19.	179
4-1	Recursive network structure.	184
5-1	Cost of automatic program generation.	198
5-2	COBOL cost factors for 1000 source lines.	200
5-3	$ Cost for 1000 lines of COBOL in 1974.	203
5-4	Cost of generating a design.	206

1 THEORY AND SUPPORT.

1.1 Introduction.

Data management programmers are finding their jobs are getting tougher because of the gradual replacement of sequential data bases by network data bases. In addition, there is a new job called "Data Administrator" [CODASYL 1971a] for handling the data structure problems associated with network data bases. The goal of this thesis is reduction of these data management tasks by developing and applying a practical theory of data structure. To insure the practical flavor of this research, I have selected the Data Base Task Group (DBTG) [CODASYL 1971a] report as the specification of the data management system in which the applications function.

An implemented system that automates both information retrieval and data base design demonstrates the application of the theory. To do this, the theory is captured in a Frame, a set of formal rules in the logic of programs. An Automatic Programming Generator (APG) compiles this Frame to an operating program. I will discuss two Frames that resulted in two programs. One of these programs generates information retrieval programs. The other program designs a data base structure.

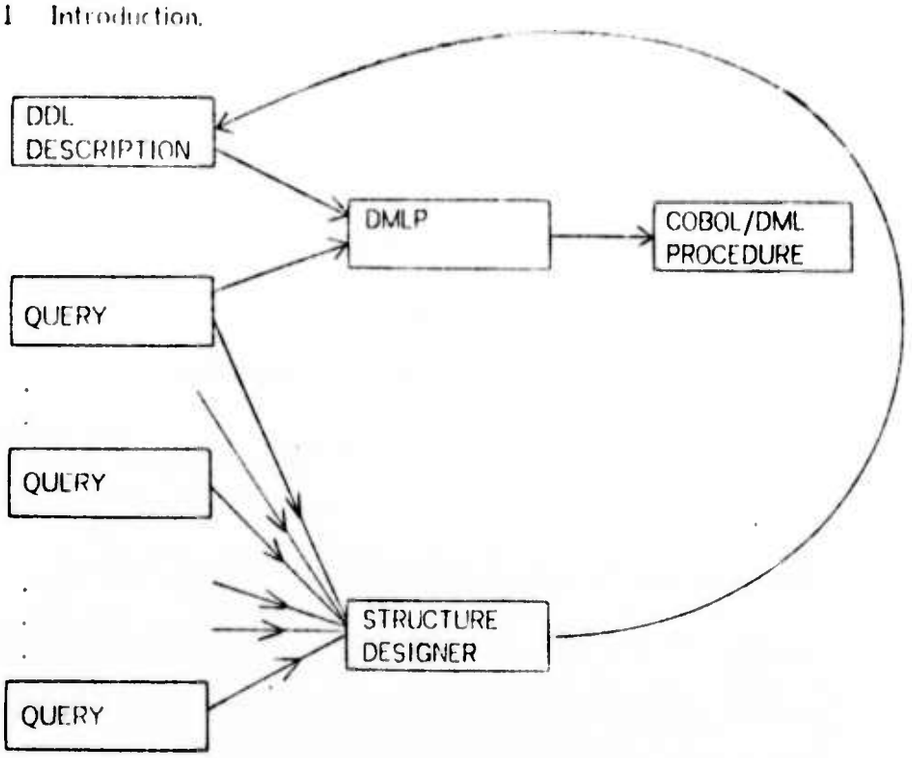


Figure 1-1. Applications of data structure theory.

Figure 1-1 illustrates these two programs, a Data Manipulation Language Programmer (DMLP) and a structure designer. The DMLP translates a query to COBOL procedure augmented by Data Manipulation Language (DML). To do this it uses a Data Definition Language (DDL) [CODASYL 1971a] description of the data base structure.

The DMLP is data base independent. The Frame that defines the DMLP contains only generat knowledge of data structures and programming. Specific knowledge of a particular data base is contained in the DDL description.

Note that the DMLP does not perform the information retrieval described in the query. It only generates a procedure that will retrieve the desired information when it is executed.

The other program in Figure 1-1 generates a DDL description from a set of queries; it tries to find a design that captures all data relationships implied by the queries.

The logical structure of the data base is of primary concern in this thesis. Actual physical mapping of the data to that structure will not be considered. Nor does the system consider certain data characteristics including data type, retrieval frequency, update volatility and data volume.

The accomplishments of this thesis are as follows:

Chapter 1: presents general concepts (theory) of network data structures.

Chapter 2:

(a) presents a Frame, a formal (axiomatic) representation of programming for DBTG data bases, and HI-IQ, an interactive query language.
(b) discusses compilation of (a) by the APG enabling it to write information retrieval procedures.

Chapter 3:

(a) presents a new algorithm for DBTG data base design including a Frame, a formal (axiomatic) representation of the design process.
(b) discusses compilation of (a) by the APG enabling it to design data structures.

Chapter 4: suggests further research including extension of the techniques used in Chapters 2 and 3 to other data management tasks.

Chapter 5: demonstrates the economic validity of the approach - a potential programming cost reduction of up to 98%.

Chapter 6: presents some remarks on the difficulties encountered in this research and the expected impact on related research.

Chapter 7: claims that this research bridges the gap between the relational and network models of data bases.

1.2 Relevant Literature.

This work is related to:

a) Automatic and/or structured programming,

b) Data bases and their implementation,

c) Non-procedural information retrieval.

The following discussion will proceed in the above sequence. Obviously there is no clean boundary between some of these fields: Category "c" bridges categories "a" and "b".

1.2.1 Automatic and structured programming.

Automatic programming systems have always used theorem provers of some sort. PROW [Waldinger and Lee 1969] is a clear example, being based on a predicate calculus theorem prover. Preceding this effort by several years was Simon's Heuristic Compiler [Simon 1961; also in Simon and Siklossy 1972] which was based on the General Problem Solver [Newell, Shaw and Simon 1960]. Although the terminology used by Simon does not imply the use of a theorem prover, it is not very difficult to view GPS as a theorem prover: The "Means-End Analysis" done by GPS is goal driven as are some theorem provers; GPS operators, which may be viewed as rules of inference, achieve the goal by removing differences.

Hewitt [Hewitt 1971] proposed a programming language called PLANNER in which

the rules for proving theorems (or achieving goals) may be expressed as part of the program. This facilitates the implementation of theorem provers in other problem domains (such as automatic programming or natural language understanding) because it removes a level of interpretation. Hewitt's ideas were implemented in a programming language called Micro-Planner [Sussman and Winograd 1972].

Hoare's development of a system of logic for the definition of the semantics of programming languages [Hoare 1969] has proven useful for automatic programming. So has the concept of structured programming [Dahl, Dijkstra and Hoare 1972]. The logical basis of the automatic programming system reported in [Buchanan and Luckham 1974; Buchanan 1974] was described using the Hoare logic. This system contains a compiler that translates a non-procedural definition of a programming environment (whose elements correspond in form to statements in Hoare's logic of programs) into Micro-Planner theorems and LISP functions.

A program resulting from such a compilation is a program generator that is capable of generating specific programs satisfying given sets of input-output assertions. The program generator of Chapter Two and the data base designer of Chapter Three were formed by an extended version of the aforementioned system.

1.2.2 Data bases and their implementation.

The retrieval programs that can be generated are capable of retrieving information from data bases which allow the storage of highly inter-related data. Such data bases are typically called network, hierarchical, or relational data bases, and they are quite different from the set of sequential files that traditionally formed a data base.

There exists some disagreement [Codd and Date 1974; Date and Codd 1974; Jardine 1974] regarding differences/similarities of the network and relational models of data bases. Some comments relevant to this discussion and the impact of this thesis on the disagreement are presented in Chapter Seven. It is generally agreed, however, that hierarchical data bases are a subclass of the network type.

The network data base was commercially developed at General Electric [Bachman and Williams 1964; Bachman 1965; General Electric 1965]. A specific notation for describing the structure of a network data base has been developed [Bachman 1969]. An attempt at standardization of this type of data base was made by the Data Base Task Group [CODASYL 1971a]. Several commercially available implementations of this standardized design are available, for example [Xerox 1970]. A generalized hierarchical data base system has been implemented at Bell Laboratories [Gibson and Stockhausen 1973].

In a network data base such as the DBTG specification, the relations between data must be explicitly named and defined. The DBTG terminology for a relation is a SET. The elements of a SET are linked, hence the "network" characteristic of such data bases. Codd [Codd 1970, 1972a, 1972b] and Childs [Childs 1968] have developed what is termed the relational model of data bases. In this model a relation is a defined template of related data elements. The user of a relational data base defines elements and templates rather than elements and the links between them as must be defined for a DBTG type data base.

A discussion of other data base implementations can be found in [CODASYL 1971b]. Some of the systems discussed in the CODASYL report utilize sequential or simple hierarchical data bases that do not present many of the complexities encountered with more structured data bases. Information retrieval systems based on the relational model have been implemented at IBM [Boyce, Chamberlin, King and Hammer 1973].

1.2.3 Non-procedural information retrieval.

"Non-procedural", "problem statement", and "goal oriented" languages are often discussed for information retrieval because, it is hoped, such a language will allow the information user direct access to the data without the intervention of a programmer to translate his needs to a computer procedure (e.g. the retrieval program). Many such non-procedural languages have been developed and some that are more well known are discussed in [CODASYL 1971b] and in

[Teichroew 1970]. Of the languages discussed in these reports the ones that are non-procedural are associated with the simpler, sequential data base structures.

For example, MARK IV [Informatics 1969; Postley 1969], probably the most commercially successful of the systems that specify a non-procedural language, is a forms oriented system that retrieves information based on a specification of the desired report. This system is limited to the sequential processing of files and does not allow inter-record structures although intra-record hierarchies are allowed.

As for languages for more complex data bases, Early [Early 1972] discusses the idea of language stratification. That is, a retrieval can be specified in machine language, access path language, or a relational language. Using Early's terminology, III-IQ is a relational language, and the DMLP is a translator from a relational language to an access path language (COBOL/DML). Further translation from the access path language to a machine language is accomplished by the DBTG implementation, usually via subroutine calls to a runtime library or with macro expansion during compilation.

The problem of translating from a non-procedural to a procedural language has also been discussed by Teichroew [Teichroew and Sayani 1971]. An example of such a translator for a hierarchical system was implemented at Bell Laboratories

[Pueschling and Roberts 1973].

The idea of processing a network by attacking a single hierarchy at a time comes from the work of Lavallee and others [Lavallee, Ohayon and Sauvain undated]. Their work also introduces the important concept of confluent hierarchies, the building blocks of network data bases.

1.2.4 Data base design.

[Teichroew and Sayani 1971] briefly discuss the goal of automating the data base design task. With the exception of the program discussed in Chapter 3, little progress to date has been reported.

1.3 DBTG Data Bases.

The CODASYL Data Base Task Group [CODASYL 1971a] specified a direct access data base management system of the network type. This is a so called "general" system, the idea being that many different specific data management applications will be implemented that make use of the general system. General systems very similar to the DBTG specification have been implemented by Honeywell Information Systems, called Integrated Data Store (IDS) [General Electric 1965], and by Xerox, called Data Management System (DMS) [Xerox 1970].

The DBTG report specifies two user interfaces. One is the Data Definition Language (DDL) whereby the data base structure is described in a Schema; the other is the Data Manipulation Language (DML) whereby actual data access and storage is achieved. The schema functions as a map for the data management library routines. These routines are called by the DML statements in the application program.

DBTG data base structure is defined in terms of "records", "items", "sets", "groups", and "areas". Records consist of a concatenation of items. The set is a relation between records. A set always has a unique owner record and one or more member records. A set defines a one-to-many relationship in the data base between the owner record and member records. A particular set

occurrence in the data base includes at most one occurrence of the owner record. A group is a subdivision of a record, i.e. a smaller concatenation of items. An area is a subdivision of the data base.

Record access in a DBTG system occurs in one of three ways; direct, calculated, or via a set relationship. A record can be directly accessed if its storage address is known. Calculated access is possible if the values of those items from which the storage address can be calculated are known. Records for which calculated access is desired must be so specified when the records are defined in the Schema. The items that are to be used in the calculation must also be defined in the Schema. Set relationship accesses allow the finding of an owner of a set previously accessed via one of its members or the finding of the first, last, next, or prior member of a set previously accessed.

Data Structure (DS) diagrams illustrate the structure of DBTG data bases [Bachman 1969]. In these diagrams, boxes are used to represent records and arrows represent sets. The arrow always emanates from the owner of the set and points to the set member. For example, Figure 1-2 illustrates a simple data base structure defined with two record types and one set. Although this data base can contain many DOCTOR records, each TREATING set occurrence will contain exactly one DOCTOR record. On the other hand, the set occurrence may contain many PATIENT records, one record for each patient being treated by the doctor specified in the DOCTOR record.

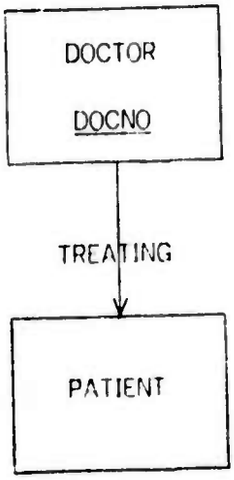


Figure 1-2. Simple DOCTOR-PATIENT data base.

In these diagrams, underlined items represent calculated keys. Therefore a DOCTOR record can be retrieved using a DOCNO value.

The data base structure in Figure 1-2 does not allow more than one doctor per patient unless patient records are duplicated. This problem does not occur with the data base of Figure 1-3.

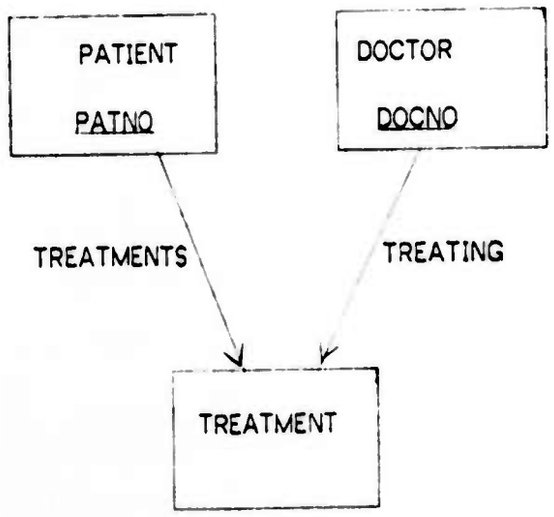


Figure 1-3. Better DOCTOR-PATIENT data base.

This structure, called a confluent hierarchy [Lavalle et al undated], captures a two-way hierarchy. In Figure 1-3 a patient may have many associated treatments and therefore many associated doctors, and a doctor may be performing many treatments, each with an associated patient.

The DDL description of the structure of a data base contains the same information as a data structure diagram.

A programmer uses DML commands to move through the data base. For example, to find the doctor with identifying number 100, the program would contain the following COBOL and DML statements (for the data base of Figure 1-2):

```
MOVE 100 TO DOCNO.
FIND DOCTOR RECORD.
```

Executing the statement

FIND FIRST PATIENT RECORD OF TREATING SET

will locate a patient being treated by doctor 100, if it is executed following the statement that found the doctor. The statement

FIND NEXT PATIENT RECORD OF TREATING SET

can then be repeatedly executed to find all of the patients of doctor 100.

The user program must check an error code to determine when all patient records in the set have been found.

Notice that there is a context which effects communication between the program and the DBTG system. The DBTG report calls the context "current". When the current owner of the TREATING set is doctor 100, then all commands to find patients in the TREATING set will find only patients of doctor 100.

1.4 Matrices, Hierarchies, Relationships and Networks.

This section presents the theory on which valid data base programming must be based. The knowledge needed by a programmer to understand data structures and their transformations is set forth. These data structure transformations can occur whenever data moves into and out of the data base.

The assumption is made that people manipulate information in either matrix or hierarchical form (or a combination of the two). This assumption may not be correct in all cases but within the realm of tabular reports, forms, etc (e.g. a business environment), this assumption holds for a large portion of the information transfers that occur. Prose reports also follow a hierarchical structure [Shefter 1958]. Further evidence of this hierarchical structuring is the decimal numbering of paragraphs in technical documents (such as this one).

A simple tabular report (Figure 1-4) can be viewed as a matrix (e.g. each line is a row vector and each column is a column vector). Typically, each column vector can be associated with an attribute whereas each row vector is associated with some type of simultaneous occurence of values for the several attributes. Frequently such simultaneous value occurences are called records. In both reports the attributes (columns) have been labeled. The simultaneous value occurences in the first report might be "patients", those in the second report "doctors".

NAME	AGE	DIAGNOSIS
SMITH	48	BOTULISM
JONES	22	APPENDICITIS
WILLIAMSON	31	MISCARRIAGE

NAME	AGE	SPECIALTY
FREDERICKS	41	G.P.
BROWN	36	HEART
SLENDER	52	GEN SURGERY
BLUE	49	GYNECOLOGY

Figure 1-4. Simple tabular reports.

Note that these simple tabular (matrix) reports can be (and frequently are) combined into hierarchically organized reports as in Figure 1-5. The hierarchical report contains additional information about relationships between records, information not available from either report in Figure 1-4. General purpose systems (such as MARK IV [Informatics 1969], RPG or COBOL Report Writer) usually allow several hierarchical levels. MARK IV, for example, allows up to nine levels.

If one thinks of a hierarchy as an upside-down tree (with the repeating records forming the branches), then the notions of above and below can be used for describing the hierarchical relationships between record types. For example, in the hierarchy present in the report in Figure 1-5, the doctor records are hierarchically above the patient records.

FREDERICKS	41	G.P.
SMITH	48	BOTULISM
JONES	22	APPENDICITIS
BROWN	36	HEART
SMITH	48	BOTULISM
SLENDER	52	GEN SURGERY
JONES	22	APPENDICITIS
BLUE	49	GYNECOLOGY
WILLIAMSON	31	MISCARRIAGE

Figure 1-5 Hierarchical report.

Once the existence of a hierarchy is recognized it becomes possible and frequently useful to calculate statistics based on a hierarchy. Examples applicable to the hierarchy illustrated in Figure 1-5 are "number of patients per doctor", "average patient age per doctor", "maximum patient age per doctor", etc. These statistics all become attributes of doctor records; e.g., the average patient age attribute for doctor Fredericks has a value of 35. Notice that in a certain sense these statistics migrate upward in the hierarchy. The information from which they are calculated is located within a set of patient records, yet the calculated value becomes an attribute of the doctor record that is hierarchically above the patient record.

By introducing the concept of a universal record that is above all other records in any hierarchy, it becomes possible to associate with the universal record all

universal attributes such as the total number of doctors (which in Figure 1-5 has a value of four), the average doctor age, and the average patient age.

Attributes also migrate downward. For example, the doctor name is also a patient attribute; e.g., the patient Williamson has the attribute that her doctor's name is Blue. The concept of downward migration is an important one to any automatic system because it reduces the amount of knowledge about the structure that a user needs to have. A system that understands downward migration can allow one user to assume that the patient record contains the doctor's name while another user can assume a hierarchical relationship.

It is also possible to <u>invert</u> a hierarchy. An example of the result of an inversion is given in Figure 1-6, where the hierarchy of Figure 1-5 has been inverted. Notice that the informational content of the reports in Figures 1-5 and 1-6 is identical (assuming that patients and doctors are uniquely identified by their names), yet the usefulness of the reports (to a particular user) is strikingly different. Similarly, the index to a book or document is an inversion of the prose it accompanies.

The confluent hierarchy discussed in section 1.3 is the structure used in a network data base to simultaneously store a hierarchy in both inverted and original form.

SMITH	48	BOTULISM
FREDERICKS	41	G.P.
BROWN	36	HEART
JONES	22	APPENDICITIS
FREDERICKS	41	G.P.
SLENDER	52	GEN SURGERY
WILLIAMSON	31	MISCARRIAGE
BLUE	49	GYNECOLOGY

Figure 1-6. Hierarchical report (inverted from Figure 1-5).

2 DATA MANIPULATION LANGUAGE PROGRAMMER.

2.1 Introduction.

This chapter will describe a computer program, called the Data Manipulation Language Programmer (DMLP), that can generate a COBOL Procedure Division corresponding to an information retrieval query. The target data bases for these queries must be of the type specified by the Data Base Task Group (DBTG) [CODASYL 1971a]. The generated Procedure Division contains the usual COBOL [CODASYL 1970] statements augmented by data manipulation statements as defined by the DBTG. The DMLP does not generate a Data Division. This task has been left to the sub-schema processor as defined by the DBTG.

A Frame (a set of formal logical rules) defines the DMLP. An Automatic Programming Generator (APG) [Buchanan and Luckham 1974; Buchanan 1974] translates this Frame to an operating program.

The DMLP uses two inputs to direct and control program generation. One of these inputs is a query whereby the user specifies the information desired. The other is a description of the data base structure: The DMLP can program for many different data bases. An example of a simple query and the resulting procedure is given in the next section.

2.1.1 Introductory example.

Figure 2-1 illustrates the query (written in the HI-IQ language) corresponding to

the command, "Please display the order number, order date, shipping date, customer number and customer name for any order specified." This query was entered interactively, and for clarity I have underlined the system prompts. (A further explanation of HI-IQ is given in Section 2.4.) Figure 2-1 also illustrates the data structure diagram for the relevant portion of the data base and the Procedure Division that was generated in response to the query and the data base description.

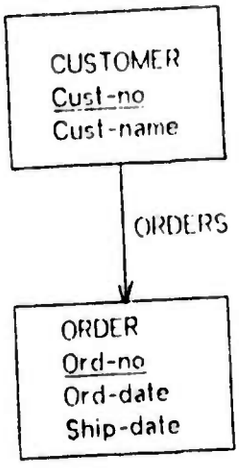
UNDERSTANDING DATA STRUCTURES

2.1 Introduction.

Inputs: (query)

```
ENTER PROGRAM NAME DEMO
READ DSK:? T
          PRIMARY RECORD FOR MAIN
          *ORDER
          CONDITIONS FOR RETRIEVAL
          *(ORD NO EQ RUNTIME)
          *NIL
          ITEMS OR STATS TO BE DISPLAYED
          *ORD NO
          *ORD DATE
          *SHIP-DATE
          *CUST-NO
          *CUST-NAME
          *NIL
POSSIBLE PORTS ARE:
(ORD-NO)
SELECT ONE OR TYPE NIL  ORD-NO
```

(data structure)



Output:

```
PROCEDURE DIVISION.
PROC1 SECTION.
PARA-100.
    OPEN AREA MAINA .
    DISPLAY 'ORD NO' 'EQ?' .
    ACCEPT ORD-NO .
    FIND ORDER RECORD .
    IF ERRORSTATUS IS NOT EQUAL TO 0  NEXT SENTENCE
        ELSE PERFORM PARA-101.
    CLOSE AREA MAINA .
    STOP .
 PARA-101.
    GET ORDER RECORD .
    FIND OWNER RECORD OF ORDERS SET .
    GET CUSTOMER RECORD .
    DISPLAY ORD-NO ORD-DATE SHIP-DATE CUST-NO
        CUST-NAME .
```

Figure 2-1. Simple query with generated procedure.

At the end of the query the system has indicated that ORD-NO is a possible port. In this query, order number qualifies as a port for two reasons: (a) It has been defined as a calculated key in the data base description. (b) Order number is used in an equality test in the query. It is therefore possible to directly retrieve the order record, given the value that order number must have.

The user may accept, reject or select a port since information necessary for this decision is not held by the system. The generated program uses the port to enter the data base. In other words, the port is the start of the access path.

This query is not complete: The particular order number for the order that is to be retrieved has not been specified. The keyword RUNTIME means that a value must be provided when the generated program is executed. The resulting program is interactive and asks the user "ORD-NO EQ?" to complete the query. The FIND is then possible because the calculated key, ORD-NO, has a value through execution of the ACCEPT statement. Following the FIND, ERRORSTATUS is checked to see if the FIND was successfully completed.

In PARA-101 the program GETs the appropriate items in core and DISPLAYs them. Note that the system has determined from the data base definition that not all desired items are contained in the ORDER record, in fact CUST-NO and CUST-NAME are contained in the CUSTOMER record. By the principle of downward migration, the system permits the user to proceed as if CUST-NO and

CUST-NAME are part of the ORDER record because the system knows that unique values for these items can be determined for every ORDER record occurence. The constructed program contains a FIND OWNER statement so that these values can be retrieved and displayed.

2.1.2 Extended capability.

The preceding example was extremely simplified for expository purposes. Additional features of the system, not brought out by the example, are as follows:

Specification of complex retrieval conditions containing conjunction, disjunction, and universal and existential quantification.

Specification of conditional output for exception reporting.

Complex nested hierarchical retrieval and output descriptions.

Calculation of totals, averages, counts, minima, and maxima.

Rules for efficient code generation.

There are many possible extensions to the system which are described separately in Chapter Four. These extensions serve as a good indication of the present limitations and shortcomings of the system.

2.1.3 By-products.

The effort of creating the DMLP yielded some interesting by-products:

The HI-IQ query language especially suited for network data bases.

A Backus-Naur Form [Naur et al 1960] description of an entire class of programs.

An Algol-to-COBOL conversion program.

A separate section of this chapter will describe each by-product.

2.1.4 System overview.

The DMLP is composed of two major subprograms as illustrated in Figure 2-2. The Request Handler interacts with the user as he enters his query and reports any differences between the query structure and the data base structure. The Request Handler has knowledge of the data base structure and some general knowledge of how the program generator will function. It generates a set of assertions describing the query. These assertions will direct and constrain the program generation phase. After an entire query has been processed by the Request Handler it poses the goal, "write a program", to the Program Writer.

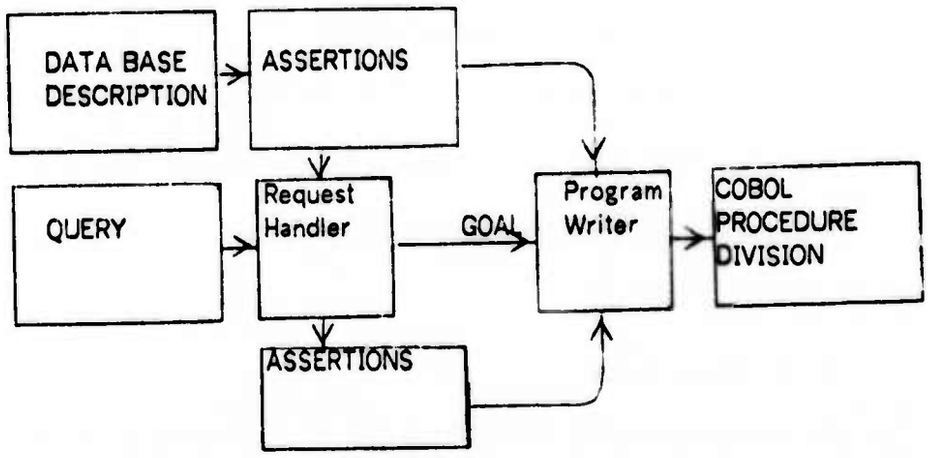


Figure 2-2. System flows.

2.1 Introduction.

The Program Writer attempts to satisfy the goal posed by the Request Handler using:

Programming techniques defined in the system.

Assertions describing the target data base structure.

Assertions describing the information retrieval query.

2.1.5 Chapter outline

To aid the reader in the necessary jumping around between the sections so that he can "bootstrap" his understanding, an outline of the sections is presented below. Not essential to an understanding of the system are sections 2.3, 2.9 and 2.10.

2.1 Introduction.
Presents a chapter outline, a simple example and the top level system structure.

2.2 The APG System.
Describes the logical basis of the axiomatic representation, and the program generation method in general.

2.3 Program Generation vs a Generalized Interpreter.
A justification of the approach, is relatively independent of the other sections.

2.4 HI-IQ, the Query Language.
Describes the primary input to the system.

2.5 Request Handler Assertions.
Describes the internal description of a query, used to direct and constrain program generation.

2.6 A BNF Description of the Generated Procedure.
Describes the system output in Backus-Naur form, useful for discussion of the types of program constructs generated.

2.1 Introduction.

2.7 Assertions.
Describes the simple Boolean expressions with which more complex statements of the logic can be made.

2.8 A Frame for the Semantics of Data Structures.
Describes the rules, stated in terms of the assertions, whereby the system "understands" data structure and Data Manipulation Language.

2.9 Efficiency Considerations.
Describes what was done, and not done, to insure generation of efficient programs.

2.10 Algol to COBOL Conversion.
Describes translation to COBOL from an internal Algol-like representation of the completed program.

2.11 Examples of Procedure Generation.

2.2 The APG System

Program construction is carried out using a domain independent automatic program generation system, hereafter denoted by APG, reported in [Buchanan and Luckham 1974; Buchanan 1974]. The APG has been extended in form as well as content for the purposes of this thesis. To sketch the logical basis of the APG, I review some elements of the logic of programs and show how the descriptive formalism for APG (called a Frame) is formulated and used in program generation. Sections 2.2.1 and 2.2.2 have been condensed directly from the original reports [Buchanan and Luckham 1974; Buchanan 1974; Igarashi, London and Luckham 1973; Hoare 1969].

2.2.1 Logic of Programs.

Statements of the logic are of the form P{A}Q where P,Q are Boolean expressions (often called assertions) and A is a program or program part. P{A}Q means "if P is true of the input state and A halts then Q is true of the output state".

A rule of inference is a transformation rule from the conjunction of a set of statements (premises, say H1 ,...,Hn) to a statement (conclusion, say K). Such rules are denoted by

$$\frac{H1\ ,\ldots,Hn}{K}$$

2.2.2 Frames.

The rules in a frame F are of three kinds:

PROCEDURES transform states into states and are expressed as statements in the logic of programs.

SCHEMES are methods for constructing programs and are expressed as rules of inference in the logic of programs.

RELATIONAL LAWS: definitions and axioms which hold in all states and serve to "complete" incomplete state descriptions by permitting first order deduction of other elements of a state from those given.

A problem for program construction may be stated as a pair <I,G>, where I is an input assertion (or initial state) and G is the output assertion (or goal that must be true in the output state). The program construction task is to construct a program A such that I{A}I', where I'⊃G. A solution is the sequence of rules of F used in the construction of the solution program A.

Notation: Substitutions, denoted by α, do not replace any variable that occurs in the initial state I. Expressions, all of whose variables occur in the initial state are called "fully instantiated". |- denotes a first order deduction using F and the standard rules described below.

Standard rules: A set of rules representing standard programming knowledge are implemented in the program construction methods of the problem solving

algorithm:

R0. Assignment Axiom: $P(t)\{x \leftarrow t\}P(x)$

R1. Rule of Consequence:
$$\frac{P \supset Q, Q\{A\}R \quad P\{A\}Q, Q \supset R}{P\{A\}R \qquad P\{A\}R}$$

R2. Rule of Composition:
$$\frac{P\{A\}Q, Q\{B\}R}{P\{A;B\}R}$$

R3. Rule of Invariance: if $P\{A\}Q$ and $I \vdash P$ then $I\{A\}Q \cup I'$ where I' is the largest subset of I consistent with Q.

R4. Change of Variables:
$$\frac{P(x)\{A(x)\}Q(x)}{P(y)\{A(y)\}Q(y)}$$

R5. Conditional Rule:
$$\frac{P \wedge Q\{A\}R, \quad P \wedge \neg Q\{B\}R}{P\{\text{IF } Q \text{ THEN A ELSE B}\}R}$$

Frame rules: A Frame defines a programming environment using the rules described below. These rules are used in conjunction with the standard rules to generate programs.

S1. Primitive procedures (or operators): the rule defining procedure p is of the form $P\{p\}Q$. The assertions P and Q are the pre- and post-conditions of p. p must contain a procedure name and parameter list.

S2. Iterative rules: an iterative rule definition containing the Boolean expressions P(basis), Q(loop invariant), R(iteration step goal), L(control test) and G(rule goal) is a rule of inference of the form:

$$\frac{P, \vdash Q, \quad Q \wedge L\{?\}R, \quad R\{??\}Q \vee \neg L}{P\{\text{while L do } ?;??\}G}$$

S3. Definitions: A definition of G in terms of P is a logical equivalence |- P≡G.

S4. Axioms: A frame axiom P is a logical axiom |- P.

The Frame is compiled by the APG to form the DMLP. Each rule in the Frame results in a Micro-Planner theorem. Such Micro-Planner constructs are not actually theorems, but that terminology will be adhered to because of historical precedent.

Each compiled theorem contains premises (or a pre-condition) and conclusions (or a post-condition). For example, the Micro-Planner theorem corresponding to the rule P{A}R has a theorem body for the pre-condition P and a calling pattern for the post-condition R.

2.2.3 Program generation.

The theorems are used in a recursive subgoaling procedure to generate information retrieval programs. The recursive procedure first builds a plan for the target program in depth-first fashion. The plan is a tree and the branches from a node correspond to the subgoals spawned at that node. For example, if the current goal is R, and R is not directly true in the current state, then the system examines the set of theorems and selects one which has a post-condition, say Q, that matches the current goal, i.e. R⊂Qα for some substitution α. Qα may not be a fully bound formula, but a complete binding will be constructed during the generation process.

If the rule instance P₂{A}Q₂ achieves R as above, then P₂ becomes the current goal. If I{B}P₂, (I is the current state), then by the rule of composition I{B;A}Q₂, and by the rule of consequence, I{B;A}R. The system finds B;A as the program to achieve R from the initial state I.

The subgoaling process does not usually distinguish, except as noted below, between the types (S1 through S4) of Frame rules. The result is that all rules are scanned for a post-condition matching the current goal. The subgoaling process does distinguish between rules of type S4 and other rules in that only S4 rules or the set of assertions can be used to prove the pre-conditions of an S4 rule. Rules of type S3 and S4 are distinguished from the other rules in that they cannot change the set of assertions. This is a consistent distinction. Since only program segments can effect changes in the environment (when executed); only rules describing their effect should be allowed to change the state description.

Rules may be specified having a pre-condition which matches an assertion in its post-condition. Such a rule may be recursive. If it is not recursive then it may not be used to satisfy its own pre-condition.

With the APG, Buchanan also introduces an interesting improvement to the logic of programs by introducing uncertainty. The value of an assertion can be TRUE, FALSE or UNCERTAIN. This uncertain logic recognizes that there exist

assertions which can only be meaningfully tested during execution of the generated program. Use of a rule containing an uncertain assertion in its pre-condition will result in the generation of a conditional procedure.

Assertions describe the pre- and post- conditions of the rules. Assertions also describe the current state. The 58 different assertions used in the data base programming environment are described in section 2.7.

2.2.4 APG extension.

The APG constructs a program of the form P{A}Q if Q contains a complete and specific description of the goal. This is a correct but quite cumbersome approach for the data base retrieval programming problem because the goal statement is quite complex. Increased complexity in the goal statement results in increased complexity in sub-goals so that the search tree becomes quite large.

The APG paradigm has been modified for the data base application to permit specification of most of the goal in P! That is, the goal Q is the general "write a program" and P contains the specific constraints on the information set to be retrieved by the goal program. In other words, the query becomes part of the programming environment.

This allows selection of specific sub-goals from the set of assertions (P) by

those rules which are specifically directed by the subgoal in question. Filtering the goal to the appropriate rules in this manner reduces the "baggage" carried by the sub-goaler and dramatically reduces fruitless search: On average only 14% of the rules tried are unneccessary.

2.3 Program Generation vs a Generalized Interpreter.

The terms compilation, interpretation and automatic program generation are related, but I differentiate between them as follows:

> Compilation - translation of an entire program from high level procedure to low level procedure.
>
> Automatic program generation - translation from a non-procedural description of a program to procedure.
>
> Interpretation - statement by statement translation and execution of procedural commands or non-procedural descriptions.

Rather than generating reusable procedure, interpreters generate immediate results. In a production environment, the classical break even analysis shows that procedure generators have a cost advantage if expected usage exceeds the break even point (Figure 2-3).

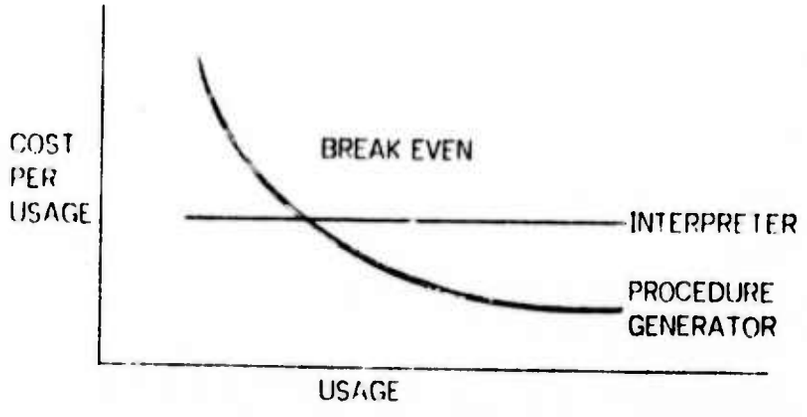


Figure 2-3. Average cost per usage for generated and interpreted programs in a production environment.

2.3 Program Generation vs a Generalized Interpreter.

A break even analysis is important in the choice of selecting a vehicle for a particular application, but in a research environment this is not the type of choice being made. Instead, researchers should attempt to provide both so that the choice is available to users.

Procedure generators are data free. This permits partial debugging of a system without a data base. To do the same with an interpreter would require the construction of a data simulator for the interpreter.

Data freeness was an important reason for building the DMLP as an automatic programming system. As a matter of fact, no data bases have been built with which to test the generated procedures. I expect that some generated programs will probably contain errors because this type of testing has not been done, but this thesis stands as a demonstration of how far the work could proceed without data bases.

Other reasons for choosing the APG were:

> The formalism discussed in section 2.2 - helps break down the problem and provides a framework for discussion.
>
> The generation of programs as a product - provides feedback to the researcher on the operation of the DMLP and can be used to discuss results.

The second advantage is not major because the same can be accomplished with an interpreter if it has an internal trace.

2.3 Program Generation vs a Generalized Interpreter.

In a preceding paragraph I stated that researchers should build both interpreters and procedure generators so that the best method can be selected in the production environment. My experience with the DMLP as a procedure generator will aid in building such an interpreter: Section 2.6 presents a BNF description for all programs corresponding to HI-IQ queries.

2.4 HI-IQ, the Query Language.

In Chapter One I stated that people tend to communicate information in hierarchical, matrix or combined hierarchical-matrix format, even if we are aware of the multi-dimensionality of the data. Usually we communicate a particular piece of information within the context of only a few relations that it may have to other information. This is so because speech, hearing, writing and reading (but perhaps not vision) are essentially serial (single channel, one dimensional) processes.

To communicate information of more than one dimension through a single channel requires the use of markers and codes to establish dimensionality contexts. After encountering several context changing markers in a communication, people soon find it difficult to keep several "trains of thought" (note the linearity implicit in this idiom) going at once.

HI-IQ (Hierarchical Interactive Query) language is designed for the specification of hierarchical-matrix reports. Other design goals for HI-IQ were flexibility in specifying hierarchical contexts, and the ability to specify the calculation of statistics. I also wanted to make the system usable for people with minimal knowledge of data base structure. This meant that the HI-IQ processor had to understand some of the concepts discussed in Chapter 1 such as downward attribute migration and confluent hierarchies.

Because of the preponderance of hierarchies in report structure, statistical calculation, and logical quantification, it seemed only natural to give HI-IQ a hierarchical structure. The hierarchical query structure is reinforced visually for the user by further indentations of system prompts for every hierarchical level referenced.

A HI-IQ query contains one or more hierarchical levels. Each level is used to specify a matrix in the output report, to specify the calculation of a statistic, or to check the truth value of a quantified condition. A simple one level query results in a report consisting of a single matrix which contains no statistics. Such a query is illustrated in Figure 2-1.

It is not unusual for the definition of a particular level to be interrupted by the definitions of lower levels. If the user desires the calculation of a particular statistic, say an average, then the system next asks him to define the calculation of that average before proceeding with the further specification of the level in which the average was requested. The prompt indentation indicates to the user which hierarchical level he is currently in.

Figure 2-4 presents a BNF description of HI-IQ. Since HI-IQ is an interactive language, only portions of a query are typed by the user. To distinguish such entries from the characters typed by the system, all system typed characters have been underlined.

Backus-Naur Form (BNF) [Naur et al 1960] is a formalism invented for the description of programming languages, specifically the grammatical structure (syntax rather than semantics) of those programming languages. BNF can also be used to describe non programming languages, such as a restricted subset of English. An excellent description and illustration of BNF can be found in [McKeeman et al 1970].

The sequence of prompts for a particular hierarchical level always consists of three subsequences. These three subsequences are <Record name>, <CONDITION LINES> and <ITEM LINES>.

```
<QUERY>              ::=  <LEVEL>

<LEVEL>              ::=  PRIMARY RECORD FOR (<COMMAND>)
                          * <Record name>
                          CONDITIONS FOR RETRIEVAL
                          <CONDITION LINES>
                          ITEMS OR STATS <MODIFIER>
                          <ITEM LINES>

<CONDITION LINES>    ::=  * NIL |
                          <CONDITION LINE> <CONDITION LINES>

<CONDITION LINE>     ::=  *OR | *ALL <LEVEL> |
                          *ANY <LEVEL> | *<TEST>

<TEST>               ::=  (<IOC> <REL> <IOCR>) |
                          (<IOC> <REL> <STAT>) <LEVEL> |
                          (<STAT> <REL> <IOCR>) <LEVEL> |
                          (<STAT> <REL> <STAT>) <LEVEL> <LEVEL>

<IOCR>               ::=  RUNTIME | <IOC>

<IOC>                ::=  <Item name> | <Constant>

<REL>                ::=  LE | LT | GE | GT | EQ | NE

<ITEM LINES>         ::=  *NIL |
                          <ITEM LINE> <ITEM LINES>

<ITEM LINE>          ::=  *<IOC> | *<STAT> <LEVEL> |
                          *REPEAT <LEVEL> | *ONE <LEVEL> |
                          *COND <TEST>

<COMMAND>            ::=  MAIN | ALL | ANY |
                          REPEAT | ONE | <STAT>

<STAT>               ::=  COUNT | TOT | AVE |
                          MIN | MAX

<MODIFIER>           ::=  TO BE DISPLAYED | FOR AVE |
                          FOR TOT | FOR MIN | FOR MAX
```

Figure 2-4. BNF for the HI-IQ language.
(System prompts are underlined.)

2.4.1 <Record name> subsequence.

The <Record name> subsequence consists of a single system prompt and user reply wherein the user must name the context record for the current hierarchical level. It is possible to have the system determine the context record for a particular level from the other two prompt subsequences. This would further reduce the knowledge that the user must have to use the system, but it would also increase the possibility of undetected errors because of the loss of redundancy.

2.4.2 <CONDITION LINES> subsequence.

The <CONDITION LINES> sub-sequence of prompts in a query level is of indefinite length. This prompt sequence defines the condition that must be true to retrieve the context record. The condition is specified using the logical connectives AND and OR and a set of tests in a disjunctive form. That is to say, if A, B, C, and D are all tests, A∧B∨C∧D is equivalent to (A∧B)∨(C∧D). However, the latter specification is not allowed; the user cannot control the bindings of the logical connectives, AND and OR. This is not a major restriction. Any condition can be specified in disjunctive form, albeit in a cumbersome way. Because of the immediate binding of AND, it is the default connector and need not be specified by the user.

Particular tests are of the form (A REL B). REL can have one of six values: EQ, NE, LT, LE, GT, GE. "A" can be an item name, a statistic, or a numeric or

non-numeric literal. "B" can be any of these; in addition "B" can be the keyword RUNTIME. The use of RUNTIME signals that the generated program will be an interactive program. If execution of the generated program becomes dependent on an actual value for "B", it (the generated program) will prompt the user with "A REL?", and the user's reply will be used to determine the truth value of (A REL B).

Universal or existential quantification can be specified as part of a condition. Since quantification is only meaningful over a set of possible values, the user must be ready to define a new hierarchical level for every quantifier specified. After encountering either of the quantifiers ALL or ANY, the system automatically proceeds to prompting for the definition of a new hierarchical level.

The system also proceeds to a new hierarchical level whenever the user specifies a statistic so that the calculation of the statistic can be defined. A statistic is specified with one of the following keywords: COUNT, TOT, AVE, MIN, and MAX.

```
CONDITIONS FOR RETRIEVAL
*(PATNO EQ RUNTIME)
*(PATAGE LT 25)
*NIL

CONDITIONS FOR RETRIEVAL
*(SALARY LT 6000)
*OR
*(SALARY LT 10000)
*ANY
   PRIMARY RECORD (ANY)
   *DEPENDENT
   CONDITONS FOR RETRIEVAL
   *(AGE LT 21)
   *NIL
*NIL
```

Figure 2-5. Two examples of retrieval conditions.

Figure 2-5 illustrates two retrieval conditions. The first is a simple conjunction of two tests. The second condition indicates that a record (employee) should be retrieved if the employee has a salary below $6,000, or if he has a salary below $10,000 and at least one dependent child.

2.4.3 <ITEM LINES> subsequence.

The <ITEM LINES> subsequence is also of indefinite length and is used to define the matrix associated with the current hierarchical level. This matrix is either an output matrix (for the report) or a statistical matrix, depending on the command which invoked the current hierarchical level. The system calculates

the statistic in each column of a statistical matrix. In other words, every column is totalled or averaged, or the minimum or maximum is found in every column of the statistical matrix.

The reply to a prompt in the <ITEM LINES> subsequence must be any one of the statistical commands, an item name, the REPEAT command, the ONE command, the COND command or NIL. NIL terminates the subsequence.

Entering an item name or COUNT defines a column of the matrix. Entering any other statistic will define one or more columns of the matrix depending in turn on the number of columns in the matrix defined for that particular statistic. This corresponds to the upward migration of statistical values that I discussed in Chapter One.

Any of the commands (with the exception of the COND command) will cause the system to initiate a new hierarchical level, so that the user can further define the action associated with the command. The REPEAT command is used for generating hierarchical reports. Figures 2-6 and 2-7 illustrate a complete query and the report it defines.

```
PRIMARY RECORD(MAIN)
*DOCTOR
CONDITIONS FOR RETRIEVAL
*NIL
ITEMS OR STATS TO BE DISPLAYED
*DOCNAME
*DOCAGE
*SPECIALTY
*REPEAT
      PRIMARY RECORD (REPEAT)
      *PATIENT
      CONDITIONS FOR RETRIEVAL
      *(PATAGE GT 21)
      *NIL
      ITEMS OR STATS TO BE DISPLAYED
      *PATNAME
      *PATAGE
      *DIAGNOSIS
      *NIL
*NIL
```

Figure 2-6. Query (P1) for the report of Figure 2-7.

FREDERICKS	41	G.P.
SMITH	48	BOTULISM
JONES	22	APPENDICITIS
BROWN	36	INTERNIST
SMITH	48	BOTULISM
SLENDER	52	GEN SURGERY
JONES	22	APPENDICITIS
BLUE	49	GYNECOLOGY
WILLIAMSON	31	MISCARRIAGE

Figure 2-7. A hierarchical-matrix report.

The top level matrix of this report contains three columns for DOCNAME, DOCAGE and SPECIALTY. The secondary matrix, hierarchically nested in the top level matrix, also contains three columns, PATNAME, PATAGE and DIAGNOSIS.

Note that in Figure 2-6, the prompt sequence for both levels of the report included all three sub-sequences as defined earlier. This query specifies a condition in the second level (on the retrieval of patients). This condition will not affect the retrieval of DOCTOR records or any other records not within the context of the PATIENT record. The condition (PATAGE GT 21) applies only to this particular context of the PATIENT record. The PATIENT record could have been referenced elsewhere in the query, and the condition (PATAGE GT 21) would not have applied.

The function of the ONE command is very similar to the REPEAT command except that only the first line of the matrix at the next level will be retrieved and displayed in the report. If the REPEAT command in Figure 2-6 is replaced with a ONE command, then the resulting report would resemble Figure 2-7 with the exception of the third line (which would not be included).

The user can control the appearance of particular attribute values on a particular line with the conditional output (COND) command. It is especially useful for exception reporting. Subsequent to encountering the COND command, the system responds as if a new hierarchical level had been specified, except that the first prompt sub-sequence is skipped. The first sub-sequence is not necessary because COND cannot change the record context.

There are a few other cases in which the full prompting sequence is not applicable, and other prompting sub-sequences will occasionally be suppressed. The third sub-sequence is not entered for the COUNT command because counting applies only to line occurrences of a matrix, the columns of the matrix do not affect it.

Similarly, it does not make sense to specify a matrix within the context of a condition quantifier (ALL or ANY), so again the third prompt sub-sequence is not entered by the system.

2.4.4 Port selection.

When the query has been completed the system tells the user which items, if any, might be used for a calculated direct access. The user is then asked to select one of these items or none. Proper selection can reduce searches through the data base. For an item to qualify for use as a calculated key it must satisfy all of several restrictions:

a) It must be defined as a calculated key.

b) It must have been used in a test with an equality relation.

c) If used in a test within a disjunct, then it must have been used in a test within every disjunct in that condition, and every such test must satisfy condition b.

The need for the first two restrictions is obvious. The third restriction is necessary because use of an item as a calculated key which meets the first two but not the third restriction cannot result in improved efficiency of the generated program. This is so since all disjuncts not containing the necessary test will cause an unlimited search anyway. An improvement for a more sophisticated system would be to relax restriction c to require a calculated key in every disjunct, but not necessarily the same calculated key.

2.5 Request Handler Assertions.

The Request Handler describes the query with a set of assertions. An assertion must conform to one of the templates in Figure 2-8.

There will be at most one ISPORT assertion for a particular query. This is asserted only if REC satisfies all criteria listed in section 2.4.4, and if it has been selected by the user for use as a calculated key.

There is exactly one TOBEOPND assertion per query. The AREAS list indicates which areas of the data base contain records that will be accessed during the processing associated with the query. Determination of the AREAS list is not simply accomplished by tallying the names of the areas that contain the records referenced in the query. It is possible that the generated program will access areas not directly referenced via record names in the query. This situation occurs if an access path between two records passes through an intermediate record. Determination of the AREAS list therefore involves a determination of all access paths.

ISPORT(REC)
Use REC as a port.

TOBEOPND(AREAS)
AREAS is a list of all areas containing records which may be accessed in the query.

LINKS(TYPE,REC1,REC2,LEVEL)
LEVEL is a Dewey-decimal identification of a query level. This level has REC2 as the context record and was entered with the command indicated by TYPE from a level which had REC1 as the context record.

FOR(COND,LEVEL)
COND specifies the condition whose truth must be established prior to any processing of the matrix.

TOBEUSED(ITEMS,LEVEL)
ITEMS identifies the columns in the matrix for LEVEL.

ISVAR(VAR)
VAR is a system generated variable.

Figure 2-8. Templates for Request Handler assertions.

LINKS, FOR and TOBEUSED are each asserted at most once for every level specified in the query. There is a one to one correspondence between these three and the three subsequences of prompting. The LINKS assertion defines the context of a query level and assigns the LEVEL identifier. The FOR assertion defines the retrieval condition, and the TOBEUSED assertion defines the matrix.

A Dewey-decimal scheme is used to identify the levels of the query. The top

level is indentified as X. The first level occuring within the context of the top level is identified as X.1. X.2.1 identifies the first hierarchy in the second hierarchy occuring within the top level of the query.

2.5.1 LINKS assertion.

The TYPE parameter of LINKS is actually a list containing two sub-parameters. Values of the first sub-parameter are limited to the names given in Figure 2-9. With the exception of MAIN, these are all commands which invoke new query levels. MAIN is used to identify the top level of the query and has the same interpretation as the REPEAT command.

The second TYPE sub-parameter is a unique system-created variable name. This variable is used for counting record occurences if the first sub-parameter is COUNT or AVE, or for controlling quantification if the first sub-parameter is ONE, ALL or ANY. Although TYPE will always contain a variable name as its second parameter, this variable is only used by the Program Writer if the first parameter in TYPE is one of the five commands indicated above.

MAIN	ALL	TOT
REPEAT	ANY	AVE
ONE	COUNT	MIN
		MAX

Figure 2-9. Possible values of the TYPE parameter in LINKS.

2.5.2 FOR assertion.

The COND parameter of FOR is a list containing all of the tests specified in the CONDITIONS FOR RETRIEVAL for a particular query level. A test is described in a sub-list containing five entries. The first entry is the relation involved in the test and the second and fourth entries are the arguments of the test. The third entry gives the query level which defines the calculation of the first argument. Similarly, the fifth entry gives the level number associated with the second argument of the assertion. These level numbers will be the same as the value of LEVEL in the FOR assertion if the argument is not to be calculated but is a constant or is available from the context record.

For example, such a list might be (EQ COUNT X.1 5 X). This test indicates that a count, as defined in the X.1 LEVEL, must be equal to 5.

Disjunction and conjunction is indicated in the COND list as follows. A simple list of tests represents a conjuction of those tests. A list in turn, of such conjunctions represents a disjunction. This list structure bears a close resemblance to the disjunctive form that the user must use to phrase the retrieval condition. If A, B, C and D are tests, then the COND list for $A \wedge B \vee C \wedge D$ would be ((A,B)(C,D).

DMLP constructs tests to enforce quantification. If the user specifies universal quantification, the system inserts the test (ALL EQ 0). The Program Writer will

eventually construct the program so that a variable associated with ALL (defined in the TYPE parameter of the LINKS assertion) is set to non-zero if the associated condition is ever false. This variable, also called a quantification flag, signals the truth value of the entire condition.

Similarly, specifying existential quantification results in a test (ANY EQ 1). The variable associated with ANY is set to non-zero in the generated procedure if the associated condition is ever true.

2.5.3 TOBEUSED assertion.

TOBEUSED describes the matrix associated with a query LEVEL. The ITEMS parameter is again a list, each entry describing a column of the matrix. The entry describing a column is in turn also a list consisting of three entries. The first of these is an item name, statistic command, or constant. The second entry indicates the query level where the calculation of the entries in the column is defined. The third entry assigns a variable name which can be used by the system for the calculation of a statistic.

As an illustration, Figure 2-10 gives the complete set of assertions derived by the Request Handler from the query of Figure 2-6. It is these assertions along with assertions describing the data base, that the Program Writer will use to generate the desired procedure.

```
TOBEOPND ((A1 A2))
LINKS ((MAIN) PORT DOCTOR X)
LINKS ((REPEAT X4) DOCTOR PATIENT (X . 1))
FOR  ((((GT PATAGE (X . 1) (21) (X . 1))))  (X . 1))
TOBEUSED  (((DOCNAME X X1) (DOCAGE X X2) (SPECIALTY X X3)
   (REPEAT (X . 1) X10))  X)
TOBEUSED  (((PATNAME (X . 1) X5) (PATAGE (X . 1) X6)
   (DIAGNOSIS (X . 1) X7))  (X . 1)))
ISVAR (X10)
ISVAR (X7)
ISVAR (X6)
ISVAR (X5)
ISVAR (X4)
ISVAR (X3)
ISVAR (X2)
ISVAR (X1)
```

Figure 2-10. Assertions describing the query of Figure 2-6.

2.6 A BNF Description of the Generated Procedure.

The Frame described in section 2.8, defines the set of programs semantically and permits a derivation of the correct syntax. This would seem to make a BNF description superfluous. However, knowing something about the general syntax of the generated programs will make the Frame easier to understand.

BNF is usually used to describe a computer language. It is possible to generate a BNF description of COBOL as augmented by the Data Manipulation Language and this BNF description would include all the programs generated by the DMLP. The BNF description in Figure 2-11 is more restricted and describes only a subset of all programs that can be written in COBOL. But it still encompasses all programs that will be generated by the DMLP.

The BNF of Figure 2-11 uses the convention that lower case strings enclosed in < > are terminal symbols. This convention is used for symbols which stand for COBOL or DMS names or literals. These names or literals must abide by the COBOL and DMS rules.

2.6 A BNF Description of the Generated Procedure.

```
<PROGRAM>           ::=  PROCEDURE DIVISION. <MAIN SECTION> |
                         <PROGRAM> <SUB SECTION>

<MAIN SECTION>      ::=  PROC1 SECTION. <MAIN PARAGRAPH> |
                         <MAIN SECTION> <SUB PARAGRAPH>

<SUB SECTION>       ::=  <SECTION NAME> SECTION. <SUB PARAGRAPH> |
                         <SUB SECTION> <SUB PARAGRAPH>

<MAIN PARAGRAPH>    ::=  <PARAGRAPH NAME>.
                         <OPEN> <PORT> <CLOSE> STOP.

<SUB PARAGRAPH>     ::=  <LOOP BODY> | <ACTION PARAGRAPH> |
                         <INTERACTIVE PARAGRAPH>

<OPEN>              ::=  OPEN AREA <AREA LIST>.

<CLOSE>             ::=  CLOSE AREA <AREA LIST>.

<AREA LIST>         ::=  <Area name> | <AREA LIST> <Area name>

<PORT>              ::=  <FIND> <ACTION>  | <LOOP CONTROL>

<FIND>              ::=  <INIT CALCKEY> FIND <Port record> RECORD.

<INIT CALCKEY>      ::=  MOVE <[illegible]S VALUE> TO <Calc key>. |
                         DISPLAY "<Calc key>" "EQ?" .
                         ACCEPT <Calc key>.

<LOOP CONTROL>      ::=  FIND FIRST <Record name> RECORD
                               OF <LOOP CONTEXT>.
                         MOVE CURRENCY STATUS FOR <LOOP CONTEXT>
                               TO <Record data base key>.
                         PERFORM <LOOP PARAGRAPH NAME>
                               UNTIL <CONTROL TEST>.

<LOOP CONTEXT>      ::=  <Set name> SET ! <Area name> AREA

<CONTROL TEST>      ::=  ERRORSTATUS IS NOT EQUAL TO 0 |
                         (ERRORSTATUS IS NOT EQUAL TO 0)
                               OR (<SYSTEM TEST>)
```

Figure 2-11 is continued on the next page.

2.6 A BNF Description of the Generated Procedure.

```
<LOOP BODY>              ::=  <LOOP PARAGRAPH NAME>.
                              MOVE CURRENCY STATUS FOR <LOOP CONTEXT>
                                   TO <VARIABLE NAME>.
                              <ACTION>
                              MOVE <VARIABLE NAME> TO <Record data base key>.
                              FIND <Record name> USING <Record data base key>.
                              FIND NEXT <Record name> RECORD
                                   OF <LOOP CONTEXT>.

<ACTION PARAGRAPH>       ::=  <ACTION PARAGRAPH NAME>. <ACTION>

<ACTION>                 ::=  <DISPLAY SEQUENCE> | <STATISTIC SEQUENCE> |
                              <CONDITION TEST> | <SET Q FLAG> |
                              <SET Q FLAG> <CONDITION TEST> |
                              <DISPLAY SEQUENCE> <SET Q FLAG> |
                              <FINDOWNER SEQUENCE> <ACTION>

<DISPLAY SEQUENCE>       ::=  <DISPLAY> | <VALUE DETERMINATIONS> <DISPLAY> |
                              <DISPLAY SEQUENCE> <CONDITION TEST> |
                              <DISPLAY SEQUENCE> <LOOP CONTROL>

<DISPLAY>                ::=  DISPLAY <DISPLAY LIST>.

<DISPLAY LIST>           ::=  <HAS VALUE> | <DISPLAY LIST> <HAS VALUE>

<STATISTIC SEQUENCE>
                         ::=  <CALCULATION> |
                              <VALUE DETERMINATION> <CALCULATION> |
                              <STATISTIC SEQUENCE> <CALCULATION> |
                              <STATISTIC SEQUENCE> <VALUE DETERMINATION>
                                   <CALCULATION>

<CALCULATION>            ::=  MOVE <COUNT VARIABLE NAME> +1
                                   TO <COUNT VARIABLE NAME>. |
                              MOVE <TOTAL VARIABLE NAME> + <ITEM OR VAR>
                                   TO <TOTAL VARIABLE NAME>. |
                              MOVE <MIN OR MAX> OF <STATISTIC VARIABLE NAME>
                                   AND <ITEM OR VAR> TO
                                   <STATISTIC VARIABLE NAME>

<MIN OR MAX>             ::=  MINIMUM | MAXIMUM

<SET Q FLAG>             ::=  MOVE <ZERO OR ONE> TO <QUANTIFICATION FLAG>.
```

Figure 2-11 is continued on the next page.

2.6 A BNF Description of the Generated Procedure.

```
<CONDITION TEST>        ::=  <IF> | <CONDITION VALUES> <IF>

<IF>                    ::=  IF <TEST> <CONTINUATION>
                                  ELSE PERFORM <ACTION PARAGRAPH NAME>.

<CONTINUATION>          ::=  NEXT SENTENCE |
                             PERFORM <ACTION PARAGRAPH NAME>

<VALUE DETERMINATIONS>
                        ::=  <VALUE DETERMINATION> |
                             <VALUE DETERMINATIONS> <VALUE DETERMINATION>

<VALUE DETERMINATION>
                        ::=  <STATISTIC CALC> | <GET> |
                             <FIND OWNER SEQUENCE> <GET> | <RUNTIME>

<STATISTIC CALC>        ::=  <STAT INIT> <LOOP CONTROL> |
                             <STAT INIT> <STAT INIT> <LOOP CONTROL>
                             MOVE <TOTAL VARIABLE NAME>
                                  DIVIDED BY <COUNT VARIABLE NAME>
                                  TO <AVE VARIABLE NAME>.

<STAT INIT>             ::=  MOVE 0 TO <STATISTIC VARIABLE NAME>.

<GET>                   ::=  GET <Record name> RECORD.

<FINDOWNER SEQUENCE>
                        ::=  FIND OWNER RECORD OF <Set name> SET. |
                             <FINDOWNER SEQUENCE>
                             FIND OWNER RECORD OF <Set name> SET.

<RUNTIME>               ::=  IF <Item name> IS NOT EQUAL
                                  TO HIGH-VALUES NEXT SENTENCE
                                  ELSE PERFORM <INTERACTIVE PARAGRAPH NAME>.

<INTERACTIVE PARAGRAPH>
                        ::=  <INTERACTIVE PARAGRAPH NAME>.
                             DISPLAY "<Item name>" "<REL>?".
                             ACCEPT <Item name>.
```

Figure 2-11 is continued on the next page.

2.6 A BNF Description of the Generated Procedure.

```
<CONDITION VALUES>      ::=  <CONDITION VALUE> |
                             <CONDITION VALUE> <CONDITION VALUE>

<CONDITION VALUE>       ::=  <FIND> | <QUANTIFICATION> |
                             <VALUE DETERMINATION>

<QUANTIFICATION>        ::=  MOVE 0 TO <QUANTIFICATION FLAG>.
                             <LOOP CONTROL>

<HAS VALUE>             ::=  <ITEM, VAR OR NUM> | <Non-numeric literal>

<ITEM, VAR OR NUM>      ::=  <ITEM OR VAR> | <Numeric literal>

<ITEM OR VAR>           ::=  <item name> | <VARIABLE NAME>

<TEST>                  ::=  ERRORSTATUS IS NOT EQUAL TO 0 |
                             <USER'S NEGATED TEST> |
                             <SYSTEM TEST>

<USER'S NEGATED TEST> ::= <HAS VALUE> <RELATION> <HAS VALUE>

<SYSTEM TEST>           ::=  <STATISTIC VARIABLE NAME> <RELATION>
                                   <ITEM, VAR OR NUM> |
                             <QUANTIFICATION FLAG> <RELATION>
                                   <ZERO OR ONE>

<RELATION>              ::=  IS EQUAL TO | IS NOT EQUAL TO |
                             IS LESS THAN | IS NOT LESS THAN |
                             IS GREATER THAN | IS NOT GREATER THAN

<REL>                   ::=  EQ | NE | GT | LT | GE | LE

<ZERO OR ONE>           ::=  0 | 1

<SECTION NAME>          ::=  PROC<Integer>

<PARAGRAPH NAME>        ::=  PARA-<Integer>

<LOOP PARAGRAPH NAME> ::= <PARAGRAPH NAME>

<ACTION PARAGRAPH NAME> ::= <PARAGRAPH NAME>

<INTERACTIVE PARAGRAPH NAME> ::= <PARAGRAPH NAME>
```

Figure 2-11 is continued on the next page.

```
<VARIABLE NAME>          ::=    X<Iinteger> | Y<Integer> | Z<Integer>

<QUALIFICATION FLAG> ::= <VARIABLE NAME>
<COUNT VARIABLE NAME> ::= <VARIABLE NAME>
<TOTAL VAIRABLE NAME> ::= <VAIRABLE NAME>
<MIN VARIABLE NAME>   ::= <VARIABLE NAME>
<MAX VARIABLE NAME>   ::= <VARIABLE NAME>
<AVE VARIABLE NAME>   ::= <VARIABLE NAME>

<STATISTIC VARIABLE NAME>
                         ::=    <COUNT VARIABLE NAME> |
                                <TOTAL VARIABLE NAME> |
                                <MIN VARIABLE NAME> |
                                <MAX VARIABLE NAME> |
                                <AVE VARIABLE NAME>
```

Figure 2-11. A BNF description of the generated procedure.

The first three productions of Figure 2-11 are applicable to all COBOL programs. These productions describe a <PROGRAM> as consisting of the statement "PROCEDURE DIVISION." followed by one or more sections. A section, in turn, begins with a statement naming the section, followed by one or more paragraphs. These three productions are perhaps somewhat different from their equivalent in a COBOL BNF description because they differentiate between the first section and all other sections and also between the first paragraph and all other paragraphs.

The need for this latter distinction is that the <MAIN PARAGRAPH> in the generated programs performs four special functions. Prior to execution of the

rest of the program it opens all necessary areas and enters the data base. At the end of execution it closes all areas opened at the beginning and stops execution.

All paragraphs other than the <MAIN PARAGRAPH> are one of three types. The first type is a paragraph that can be repetitively performed. The other two are generated for conditional execution.

Entry into the data base is done with a data base "port". A port can be a direct access record using a calculated key value, a special type of set, or a search of an entire area containing the port record. When a calculated key is used, then the <FIND> and <INIT CALCKEY> productions are used. Note the possibility of generating an interactive program in the <INIT CALCKEY> production.

If a calculated key cannot be used, the program will be constructed to contain an initial loop which either searches an area or a special type of set, known as a system set. In either case, the constructs used are described by the <LOOP CONTROL> and <LOOP BODY> productions. Note here that the <LOOP CONTEXT> production describes the differentation between loops searching a set vs those searching an area.

The <LOOP CONTROL> and <LOOP BODY> productions are two of the most important in the BNF description. <LOOP BODY> describes the construction of

the paragraph that is performed by the program section described with <LOOP CONTROL>. For every <LOOP CONTROL> there will be a unique <LOOP BODY>. This restriction cannot be indicated with the BNF because COBOL uses labels to control execution of program blocks. If retrieval programs were generated in a Algol-like language (using nesting of program blocks to control program flow) then the connection between <LOOP CONTROL> and <LOOP BODY> could be more strongly described in the BNF. I have distinguished between LOOP, ACTION, and INTERACTIVE paragraph names in the BNF to indicated these program flow connections. This difference and others between Algol and COBOL is the subject of section 2.10.

<LOOP CONTROL> and <LOOP BODY> control the code generation for every traversal from a section of code for a query level to a section of code for a lower level. The <LOOP CONTROL> and <LOOP BODY> productions correspond to the code generated for a LINKS assertion.

The <ACTION> production describes the procedure for performing the activities within a query level. These activities include matrix display, statistical calculation, quantification, and other condition testing. The action associated with quantification is the setting of the quantification flag which determines the truth of the quantification.

Another important production is the <ACTION> production. <ACTION> can occur

in a <LOOP BODY>, following a <FIND> in <PORT>, or within an <ACTION PARAGRAPH>. The code generated by <LOOP BODY> and <FIND> insures that certain records are available so that the activity can be performed. Similarly, the code generated by an <IF> production will test to see if the activity should be performed.

Since the <IF> construct is part of the <CONDITION TEST> production which in turn can be an <ACTION>, it is possible for code to be constructed which consists of an IF statement referencing a paragraph containing another IF, referencing a second paragraph containing a third IF, and so on. This is exactly the program construction that results for every conjunction of tests in a query condition. Section 2.9 explains the reason for the use of this programming style rather than one which simply reproduces the entire condition in a single IF statement.

The recursive nature of the program construction process becomes obvious when we observe that any <ACTION> can involve one or more <VALUE DETERMINATION>. <VALUE DETERMINATION> can be done with a <STATISTIC CALC> which will in turn evoke a new <LOOP CONTROL>. The general purpose of the <VALUE DETERMINATION> and <CONDITION VALUE> constructs is to generate code that will determine the values of the arguments needed by the function invoked by <ACTION>. <VALUE DETERMINATION> or <CONDITION VALUE> precede the actual display, calculation, or test processes.

```
PROC1 SECTION.
     .
     .
     IF SALARY IS NOT LESS THAN 6000
          PERFORM PARA-201
          ELSE PERFORM ..
     .
     .
PROC2 SECTION.
PARA-201.
     IF SALARY IS NOT LESS THAN 10000 NEXT SENTENCE
          ELSE PERFORM PARA-202.

PARA-202.
     MOVE 0 TO X1.
     FIND FIRST DEPENDENT RECORD OF DEPSET SET.
     MOVE CURRENCY STATUS FOR DEPSET SET TO DEPKEY.
     PERFORM PARA-203 UNTIL (ERRORSTATUS IS NOT EQUAL TO 0)
          OR (X1 IS EQUAL TO 1).
     IF X1 IS NOT EQUAL TO 1 NEXT SENTENCE
          ELSE PERFORM PARA-204.

PARA-204.
     program continuation...

PARA-203.
     MOVE CURRENCY STATUS FOR DEPSET TO Y1.
     GET DEPENDENT.
     IF AGE IS NOT LESS THAN 21 NEXT SENTENCE
          ELSE PERFORM PARA-205.
     MOVE Y1 TO DEPKEY.
     FIND DEPENDENT USING DEPKEY.
     FIND NEXT DEPENDENT RECORD OF DEPSET SET.

PARA-205.
     MOVE 1 TO X1.
```

Figure 2-12. Code generated for the condition of Figure 2-5.

Figure 2-12 illustrates the procedure that is generated for the test, "salary less

than $6,000 or salary less than $10,000 and any dependent under age 21." (The HI-IQ equivalent is illustrated in Figure 2-5.)

PROC1 in Figure 2-12 tests "salary less than $6000." If this is true then the appropriate procedure is executed. If not true, then PARA-201 is performed. PARA-201 checks the first test in the conjunction "salary less than $10,000 and the existence of any dependent child." This section of code is therefore executed only if salary is not less than $6000, in other words, only if the first disjunct of the condition fails. Code for secondary disjuncts is always located in a separate section of the Procedure Division.

PARA-201 is a simple <CONDITION TEST> containing an <IF> with no preceding value determinations. Since this paragraph will be executed within the scope of a PERFORM, NEXT SENTENCE will not cause execution of the succeeding paragraph.

PARA-202 is also a <CONDITION TEST>, but this time the <IF> is preceded by a <CONDITION VALUES> construct. In this case the <CONDITION VALUE> is a quantification so the first statement of the paragraph is a MOVE to initialize X1, the quantification flag. The remainder of the paragraph, preceding the IF statement, is a <LOOP CONTROL>. The <CONTROL TEST> for the loop contains a <SYSTEM TEST>.

PARA-203 is a <LOOP BODY> and the <ACTION> it contains is another <CONDTION TEST>. Y1 is used to store the data base key and a direct FIND is done with that data base key. The FIND using the data base key is done to recover the proper data base context in case intervening processing has also accessed the DEPSET set. It turns out that this precaution is not necessary for this particular program, but making inclusion of the precaution dependent on necessity would involve considerable backtracking during the program generation process.

PARA-205 is another <ACTION PARAGRAPH> consisting of a single <SET Q FLAG>. Upon entry to PARA-204 the entire condition is true, and PARA-204 continues with the processing specified at that level.

This terminates the example. Some additional comments regarding the BNF of Figure 2-11 are in order:

a) The principle of downward attribute migration is reflected in the <FINDOWNER SEQUENCE> as used in <VALUE DETERMINATION>.

b) The <RUNTIME> AND <INTERACTIVE PARAGRAPH> productions generate code for the interactive determination of parameter values. Note that for "ordinary" interactive value determination, the interaction occurs only once during program execution (because after being executed once, the parameter will not have HIGH-VALUES). Interaction to determine the value of a calculated key as indicated in <INIT CALCKEY>, will occur every time another direct access is desired.

c) The latter portion of the right hand side of <STATISTIC CALC> reflects the fact that an average is actually the quotient of two other statistics.

d) The use of <SYSTEM TEST> in <CONTROL TEST> helps to reduce execution time of the generated procedure by terminating loop execution if the loop was invoked to determine a value for a conditional test, and if the outcome of the test can be determined from the present value of the arguments. This pre-testing is applicable for certain statistics whose value changes monotonically during loop iterations. See also section 2.9.

e) The <DISPLAY SEQUENCE> production can invoke <LOOP CONTROL> directly in case the display of a sub-matrix was indicated in the query.

f) ERRORSTATUS is a Data Management System (DMS) variable which is set non-zero by the DMS when an error occurs or if all records in a set or area have been processed. The programs generated by the system would be better if they were able to distinguish between expected and unexpected error conditions by checking the actual value placed in ERRORSTATUS by the DMS.

g) The second alternative in the <CONTINUATION> production (invoked by <IF>) is used if the <TEST> tested in the IF statement is not part of the last disjunct in a disjunctive condition. The <ACTION PARAGRAPH NAME> paragraph will contain the code to test the other disjuncts of the condition. Such a paragraph will always be the first paragraph in a sub-section.

h) Two additional constructs for determining a value are provided if the value is to be used in a <CONDITION TEST>, e.g. as a <CONDITION VALUE>. A direct access is allowed (to force the condition to be true) if the test contains an equality relation involving a calculated key. <QUANTIFICATION> will be invoked if the test involved was constructed by the Request Handler for checking quantification. <QUANTIFICATION> results in the generation of code which determines the value of the quantification flag. The code so generated is similar to the code generated for the calculation of a statistic.

i) <SET Q FLAG> can be used several ways in an <ACTION>. If <SET Q FLAG> by itself comprises the action, then it will be the only statement in an action paragraph which is conditionally executed. <SET Q FLAG> will precede a <CONDITION TEST> only if that test is universally qualified.

The flag is set false preceding the test so that it indicates false if the test fails. If the test succeeds, the flag is reset to true. <SET Q FLAG> following a <DISPLAY SEQUENCE> occurs only for displays in the scope of a ONE command.

2.7 Assertions.

A Frame consists of a set of logical statements or rules. These rules are of four different types as discussed in Section 2.2.2 and reviewed here:

S1 Primitive procedure rule.
S2 Iterative rule.
S3 Definition rule.
S4 Axiom.

Rules and the current state are expressed with assertions. Each assertion must correspond to a template with a semantic interpretation. For example, the assertion CONTAINS(PATIENT,PATNO) states that the PATIENT record contains the PATNO item.

Evaluation of an assertion determines if it is true or false in one of several ways:

(a) if previously stated to be true or false (i.e. true or false in the current state);

(b) by evaluating a rule which has the assertion in a post-condition

(c) by evaluating a LISP function

Assertions may contain variables which can become bound when the assertion is evaluated. If CONTAINS(PATIENT,PATNO) is true in the current state, then evaluation of CONTAINS(recx,PATNO) will bind recx (a variable) to PATIENT.

DMLP uses 58 different types of assertions. Six of these were described in

Figure 2-8. Figure 2-13 describes the five used to define the database structure. An earlier paper [Gerritsen 1974] illustrates the ease of conversion from a Data Definition Language specification of a data base to a set of assertions.

INAREA(RECORD,AREA)
RECORD is contained in AREA.

CONTAINS(RECORD,ITEM)
ITEM is is contained in RECORD.

DBKEY(RECORD,ITEM)
ITEM is a data base key for RECORD.

CALCKEY(RECORD,ITEM)
ITEM is a direct access attribute (calculated key) of RECORD.

HIERARCHYGROUP(RECORD1,RECORD2,SET)
RECORD1 is the owner of SET, and RECORD2 is a member of SET.

Figure 2-13. Assertions used to describe the data base.

2.7.1 Assertions for the S1 rules.

Figure 2-14 contains the assertions which describe the results of single program steps. These assertions occur as post-conditions of rules of type S1. Those parameters which are underlined in Figure 2-14 have a uniqueness property. For example, the system will insure that a particular ITEM will contain only one VALUE: If the assertion C(X1,0) has been made followed by a later assertion

C(X1,1), then the system will erase the first assertion.

OPENED(AREAS)
AREAS is a list of areas that have been opened.

CLOSED(AREAS)
AREAS is a list of areas that have been closed.

STOPPED(NAME)
The program NAME has been stopped.

ACCEPT(VARIABLE,ITEM,RELATION)
VARIABLE contains the value entered by the user in response to the prompt "ITEM RELATION?".

CURRENT(RECORD,LEVEL)
RECORD is current at LEVEL. e.g., The named record has been found within the program segment associated with the level identifier.

INCORE(RECORD,LEVEL)
The named RECORD is in core and available for processing to the program segment associated with LEVEL.

C(ITEM,VALUE)
The named ITEM contains the given VALUE.

ANYOUTPUT(ITEMS,LEVEL)
ITEMS is a list of the columns of a matrix (that has been output at the given level).

FOUNDOWNER(RECORD1,RECORD2,SET,LEVEL)
The named RECORD1 has been found via the SET using RECORD2, a member of that SET and both records are now current at the LEVEL specified.

FOUNDNEXT(TYPE,RECORD,UNIT,LEVEL)
The next RECORD of the specified UNIT which is either an Area or Set as specified in TYPE has been found and is current for the LEVEL specified

---

Figure 2-14 is continued on the next page.

FOUND(RECORD,ITEM,VALUE,LEVEL)
The named RECORD has been found using ITEM as a calculated key with the given VALUE such that RECORD is current for the specified LEVEL.

FOUNDUSING(RECORD,KEY,LEVEL)
The named RECORD has been found using the data base KEY and is current for the specified LEVEL.

FOUNDFIRST(TYPE,RECORD,UNIT,LEVEL)
The named RECORD has been found as the first record of the specified UNIT which is either a Set or Area as specified by the value of TYPE. The record is current for the specified LEVEL.

Figure 2-14. Assertions which indicate the results of single program statements.

There is a semantic redundancy in each of the FOUND... assertions and the CURRENT assertion since a found record must be current. It would be possible to eliminate this redundancy with a rewriting of the rules, and such a rewrite would lead to a more concise system. On the other hand, the redundancy serves to guard against improper program generation if (for example) a particular rule has an incomplete pre-condition.

The assertions in Figure 2-14 appear to describe the status of an executing program. The descriptions are more properly interpreted for program generation if each is read as if preceded with the phrase "Code has been generated such that...".

ISITEM(ITEM)
ITEM is contained within some record as an attribute or data base key.

BCA*(COMMAND)
COMMAND has the value "COUNT"" or "AVE".

BTMMA*(COMMAND)
COMMAND has the value "TOT", "MIN", "MAX" or "AVE".

=(A,B)
A is equal to B.

EQ*(A,B)
A is equal to B. This assertion differs from the preceeding one in that its value can be uncertain if, for example, either A or B are program variables.

TEST(CONDITION,LEVEL)
The CONDITION is true for the program segment defined for LEVEL. CONDITION is a list consisting of a relation and two arguments. When the arguments are program variables, TEST will have an uncertain value.

Figure 2-15. Other assertions.

2.7.2 Miscellaneous assertions.

Figure 2-15 contains a set of assertions which are difficult to classify. The first is directly derivable from the assertions describing the data base structure. The next three are used to test the values of their arguments, and the last two are used to test values or insert code to test values.

2.7.3 Assertions for the S3 rules.

Figure 2-16 contains the remaining assertions. These can best be classified as

describing the general status of the program or of particular program blocks as they are constructed. This is different than the assertions in Figure 2-14 which describe the result of single program statements. All of the assertions of Figure 2-16 are post-conditions of type S3 rules. S3 rules serve to insure the proper composition of individual operators or other program blocks into larger program blocks.

ALLFORFULINST(ACTION,CONDITION,LEVEL)

The entire program segment for the given LEVEL has been constructed. ACTION indicates the command which established the purpose of the LEVEL and CONDITION contains the COND from the appropriate FOR assertion which must be true prior to the processing for ACTION.

ALLFOR(ACTION,CONDITION,LEVEL,DUMMY)

CONDITION contains one of the disjuncts, itself a conjunction of tests, from the original COND list. This assertion states that the program segment to test the conjunction has been generated. DUMMY is a kludge variable which is kept unbound to force the proper construction of disjunctive tests.

GETPORT(RECORD)

The named record has been used to establish entry into the data base.

PROGRAM(NAME)

The named program has been written. This is the goal posed by the Request Handler for the Program Writer.

DOACTION(ACTION,LEVEL)

Code for processing the named ACTION associated with the indicated LEVEL has been generated.

LINKED(ACTION,RECORD1,RECORD2,LEVEL,ARG1,ARG2)

Code for linking from RECORD1 to RECORD2 has been generated, as well as all associated processing for the prescribed ACTION. This program segment is identified by LEVEL. ARG1 and ARG2 are used to bind the values of arguments that can be used for early termination of loops. ARG1 and ARG2 are therefore associated with ACTION.

UPLINKED(RECORD1,RECORD2,LEVEL)

RECORD2 has been made current at the given LEVEL using only FIND OWNER operators to do so, starting from RECORD1.

Figure 2-16 is continued on the next page.

DETVAL(ITEM,LEVEL,ARG1,ARG2,RELATION)
The value for the named ITEM has been determined with the program segment identified by LEVEL, and the value has been stored in ARG1. ARG2 and RELATION are used to indicate the test in which the ITEM is used, if it is used in a test. The latter two parameters are used because value determination is sometimes dependent on the anticipated use of the item.

GETRUNIT(VARIABLE,ITEM,RELATION,)
VARIABLE contains the value entered by the user in response to the prompt "ITEM RELATION?" This assertion is the equivalent of ACCEPT in Figure 2-14. This duplication is another kludge to effect the proper insertion of conditional tests.

Figure 2-16. Assertions used to describe program blocks.

2.7.4 LISP functions.

Some of the assertions used in the preconditions of rules are not used in post conditions of other rules or in the current state. Such assertions may be evaluated by LISP functions. Assertions for which there exists a LISP function have ** as the last two characters in their names. These assertions and their meanings are listed in Figure 2-17.

DETALLVAL###(ITMS,ACTION)
Code has been generated to delermine the values, relative to ACTION, of all items in the ITMS list.

NEXTLEVOUT###(ITMS)
Code has been generated to process all submatrices indicated by REPEAT or ONE commands in the ITMS list.

INITVARS###(VARLIST)
Code has been generated to initialize all variables on the VARLIST list to zero.

DIVVARS###(DIVISOR,ITEMS)
Code has been generated to divide every variable in the ITEMS list by the DIVISOR, e.g. to calculate averages.

STAT###(COMMAND))
COMMAND is one of the statistical commands.

RETQ###(COMMAND)
COMMAND is a retrieval quantifier (ANY or ALL).

REPQ###(COMMAND)
COMMAND is a reporting quantifier (MAIN,REPEAT or ONE).

LITERAL###(ITEM)
ITEM is a number or non-numeric literal (enclosed in single quotes).

BINDTYP###(RECORD)
The variable TYP has been bound to the value "SET" unless RECORD has value "AREA" in which case TYP is also bound to AREA.

BINDITM###(ITEMS)
The variable ITML has been bound to a list of printable items extracted from the ITEMS list. This is done to eliminate commands which cause the printing of sub-matrices and also to replace statistic commands with the variable containing the value of the statistic.

Figure 2-17 is continued on the next page.

BINDVAL##(LIST,ITEM)
The variable VAL is bound to the second argument of an equality test contained in the LIST which has ITEM as its first argument; e.g. (ITEM EQ VAL).

READL##(ITM REL)
This assertion returns the value "ITM REL?", which is used in the generated program to prompt the user.

CONTEST##(ACTION,ARG1,ARG2)
This assertion returns a test which will be used for terminating a loop. Appropriate tests for early termination of the loop depending on the ACTION of the loop are also generated. An early termination test will involve ARG1 and ARG2.

UNCERTERRSTAT##()
This assertion always evaluates to true. However it also insures that all knowledge about the value of ERRORSTATUS becomes uncertain. This is used to indicate that the value of ERRORSTATUS becomes unknown following a data base access.

Figure 2-17. Assertions evaluated by LISP.

Of the assertions in Figure 2-17, the first four are most important as their evaluation will spawn other goals and will eventually result in additional code generation. The second four assertions (STAT## through LITERAL##) are very simple and return true or false depending on the value of their single parameter. The next three assertions (all have BIND as the first four characters) are used to bind variables to values extracted from lists. These assertions are necessary because of the list structures contained in the assertions generated by the Request Handler.

The last three assertions are unusual in that they are not evaluated for truth or failure. Instead, they return a value or change the state as is explained in Figure 2-17.

Occasionally the pre-condition of a rule will include standard LISP or Micro-Planner predicates (see [McCarthy et al 1972] and [Sussman and Winograd 1972]). These predicates are illustrated in Figure 2-18.

Finally, Figure 2-19 gives an alphabetical index of the assertions and predicates in Figures 2-8 and 2-13 thru 2-18.

THSETQ(A,B)
Sets Micro-Planner variable A to the value of B.

THASVAL(A)
True if variable A is bound to a value

NULL(A)
A is null.

CAR(A)
Returns the first element of the list A.

CDR(A)
Returns a list equivalent to A with its first element removed.

Figure 2-18. Standard LISP and Micro-Planner predicates used in the rules.

ASSERTION	FIG.	ASSERTION	FIG.
ACCEPT	2-14	GETRUNT	2-16
ALLFOR	2-16	HIERARCHYGROUP	2-13
ALLFORFULINST	2-16	INITVARS**	2-17
ANYOUTPUT	2-14	LITRAL**	2-17
BCA*	2-15	INAREA	2-13
BINDITML**	2-17	INCORE	2-14
BINDTYP**	2-17	ISITEM	2-15
BINDVAL**	2-17	ISPORT	2-8
		ISVAR	2-8
BTMMA*	2-15	LINKED	2-16
C	2-14	LINKS	2-8
CALCKEY	2-13	NEXTLEVOUT**	2-17
CAR	2-18		
CDR	2-18		
CLOSED	2-14	NULL	2-18
CONTAINS	2-13	OPENED	2-14
CONTEST**	2-17	PROGRAM	2-16
CURRENT	2-14	READL**	2-17
DBKEY	2-13	REPQ**	2-17
DETALLVAL**	2-17	RETQ**	2-17
DETVAL	2-16	STAT**	2-17
DIVVARS**	2-17	STOPPED	2-14
DOCATION	2-16	TEST	2-15
EQ*	2-15	THASVAL	2-18
FOR	2-8	THSETQ	2-18
FOUND	2-14	TOBEOPENED	2-8
FOUNDFIRST	2-14	TOBEUSED	2-8
FOUNDNEXT	2-14	UNCERTERRSAT**	2-17
FOUNDOWNER	2-14	UPLINKED	2-16
FOUNDUSING	2-14	=	2-15
GETPORT	2-16		

Figure 2-19. Index to Figures 2-8 and 2-13 through 2-18.

2.8 A Frame for the Semantics of Data Structures.

2.8.1 Rules of Type S1.

Now that all of the assertions and predicates have been described, it is possible to discuss the Frame that (in the logic of programs) forms an axiomatic representation of the semantics of data structures and DML procedure. The rules of the Frame which describe simple operators (type S1) will be discussed first. Each of these rules is of the form P{A}R where A is a single program operation or command. The post-condition R will be one or more of the assertions of Figure 2-14.

Each rule in Figure 2-20 is described with A, the operation, followed in order by P, the pre-condition, and R, the post-condition. The post-condition is further identified by the preceeding →. Variables in each rule are underscored.

Operator rules are actually fairly simple, and only two will be discussed in detail. The next to last rule of Figure 2-20, GET REC, is extremely simple and an English interpretation is:

> If REC is CURRENT in a particular program segment identified by LEVEL then executing GET REC will insure that REC is INCORE and available for processing by the particular program segment.

FIND REC RECORD is somewhat more complicated and the following interpretation will ignore some of the assertions. In English:

> If REC is a port and ITM is a calculated key and is required to be equal to VAL then either determine VAL interactively, or if VAL is not equal to RUNTIME simply move VAL to ITM

> Following the FIND, assert the record has been FOUND and is CURRENT but not INCORE. Also indicate that the value of ERRORSTATUS is uncertain.

Note the equivalence between this rule and the <FIND> production earlier discussed.

MOVE VAL TO DEST.

ISVAR(DEST)∨ISITEM(DEST)

→ C(DEST,VAL)

ACCEPT VAR.

[ISITEM(VAR)∨ISVAR(VAR)]∧
ANYOUTPUT((EV(READL** ITM REL)),VAR)

→ ACCEPT(VAR,ITM,REL)∧C(VAR,RUNTIME)

DISPLAY ITML

¬NULL.(ITML)

→ ANYOUTPUT(ITML,LEVEL)

OPEN AREAS.

TOBEOPND(AREAS)

→ OPENED(AREAS)

CLOSE AREAS.

TOBEOPND(AREAS)

→ CLOSED(AREAS)

FIND REC RECORD.

ISPORT(REC)∧CALCKEY(REC,ITM)∧FOR(FORLIST,LEVEL)∧
[THASVAL(VAL)∨BINDVAL**((EV(CAR FORLIST)),ITM)]∧
[¬=(VAL,RUNTIME)∨ACCEPT(ITM,ITM,EQ)]∧C(ITM,VAL)

→ FOUND(REC,ITM,VAL,LEVEL)∧CURRENT(REC,LEVEL)∧
¬INCORE(REC,LEVEL)∧UNCERTERRSTAT**()

FIND REC USING DBK.

DBKEY(REC,DBK)

→ FOUNDUSING(REC,DBK,LEVEL)∧CURRENT(REC,LEVEL)∧
¬INCORE(REC,LEVEL)∧UNCERTERRSTAT**()

Figure 2-20 is continued on the next page.

```
FIND FIRST REC RECORD OF UNIT TYP.
                [=(TYP,SET)∧HIERARCHYGROUP(OWN,REC,UNIT)∧
                   [=(OWN,SYSTEM)∨CURRENT(OWN,LEVELX)]]∨
                [=(TYP,AREA)∧INAREA(REC,UNIT)]

            →   FOUNDFIRST(TYP,REC,UNIT,LEVEL)∧CURRENT(REC,LEVEL)∧
                ¬INCORE(REC,LEVEL)∧UNCERTERRSTAT**()

FIND NEXT REC RECORD OF UNIT TYP.
                [=(TYP,AREA)∧INAREA(REC,UNIT)]∨
                [=(TYP,SET)∧HIERARCHYGROUP(OWN,REC,UNIT)∧
                   [=(OWN,SYSTEM)∨CURRENT(OWN,LEVELX)]]

            →   FOUNDNEXT(TYP,REC,UNIT,LEVEL)∧CURRENT(REC,LEVEL)∧
                ¬INCORE(REC,LEVEL)∧UNCERTERRSTAT**()

FIND OWNER RECORD OF ST SET.
                HIERARCHYGROUP(OWN,MEM,ST)∧¬=(OWN,SYSTEM)∧
                CURRENT(MEM,LEVELX)

            →   FOUNDOWNER(OWN,MEM,ST,LEVEL)∧CURRENT(OWN,LEVEL)∧
                ¬INCORE(OWN,LEVEL)∧UNCERTERRSTAT**()

GET REC.
                CURRENT(REC,LEVEL)

            →   INCORE(REC,LEVEL)

STOP.
                ¬STOPPED(NAM)

            →   STOPPED(NAM)
```

Figure 2-20. Operator (type S1) rules.

2.8.2 Iteration rule - type S2.

Figure 2-21 contains the single iterative rule used by the DMLP. The code generated by this rule matches that produced by <LOOP CONTROL> and <LOOP

BODY>. The rule is described with a pre-condition, loop invariant, iteration step, iteration test and a post-condition. The pre-condition and iteration test correspond to <LOOP CONTROL>. The loop invariant and iteration step is captured in the <LOOP BODY>. The variable Y1 as used in the example of Figure 2-12 is specifically introduced by the system to insure the loop invariant.

The only post-condition of the Iteration Rule is a LINKED assertion. This means that a loop is constructed for the purpose of accessing the GREC record. e.g., The goal which invokes the iteration is to find an access path from PREC to GREC and then perform the ACTION. Since a single loop traverses a single hierarchy it will establish a path from PREC to REC.

If REC is equal to GREC, the access path has been completed. If not, the rule can be recursively entered to complete the path from REC to GREC. Such recursive entry of the rule will result in nested loops. The STAYUP rule (described below) insures an end to the recursion by checking if PREC and GREC are the same, in which case the access path has been completed.

(CURSTAT,A,B,C) is a function which is expanded to "CURRENCY STATUS FOR A B".

```
PRE-CONDITION:
      BINDTYP##(PREC)^
      [[[~(PREC,AREA)vHIERARCHYGROUP(PREC,GREC,UNIT)]^
         THSETQ(REC,GREC)]vHIERARCHYGROUP(PREC,REC,UNIT)]^
      FOUNDFIRST(TYP,REC,UNIT,LEVEL)^DBKEY(REC,DBK)^
      C(DBK,(CURSTAT,UNIT,TYP,0))

INVARIANT:
      C(DBK,CURRV)

ITERATION STEP:
      LINKED(ACTION,REC,GREC,LEVEL,AR1,AR2)^
      C(DBK,(CURSTAT,UNIT,TYP,CURRV))^
      FOUNDUSING(REC,DBK,LEVEL)^FOUNDNEXT(TYP,REC,UNIT,LEVEL)

LOOP TERMINATOR:
      CONTEST##((EV(CAR ACTION)),AR1,AR2)

POST-CONDITION:
      LINKED(ACTION,PREC,GREC,LEVEL,AR1,AR2)
```

Figure 2-21. The iteration rule.

```
TOPPROG
        LINKS(ACTION,PORT,GREC,LEVEL)∧OPENED(AREAS)∧GETPORT(PREC)∧
        LINKED(ACTION,PREC,GREC,LEVEL,NIL,NIL)∧CLOSED(AREAS)∧
        STOPPED(NAM)

    →   PROGRAM(NAM)

PORTDEF
        [ISPORT(REC)∧FOUND(REC,ITM,VAL,X)]∨
        [¬ISPORT(DREC)∧LINKS(DU1,PORT,GREC,X)∧
            THSETQ(REC,SYSTEM)∧UPLINK(GREC,REC)∧]∨
        [¬ISPORT(DREC)∧THSETQ(REC,AREA)]

    →   GETPORT(REC)

STAYUP
        [[=(REC,GREC)∧CURRENT(REC,LEVEL)]∨
            [¬=(REC,SYSTEM)∧¬=(REC,AREA)∧
                UPLINKED(REC,GREC,LEVEL)]]∧
        [¬=((EV(CAR ACTION)),ALL)∨C((EV(CADR ACTION)),1)]∧
        [[¬FOR(FORLIS,LEVEL)∧DOACTION(ACTION,LEVEL)]∨
            [FOR(FORLIS,LEVEL)∧
                ALLFORFULINST(ACTION,FORLIS,LEVEL)]]

    →   LINKED(ACTION,REC,GREC,LEVEL,AR1,AR2)

UPHIERARCHY
        FOUNDOWNER(GREC,MEM,ST,LEVEL)∨
        [FOUNDOWNER(OWN,MEM,ST,LEVEL)∧UPLINKED(OWN,GREC,LEVEL)]

    →   UPLINKED(MEM,GREC,LEVEL)

ALLFORFULINSTDEF
        ALLFOR(ACTION,(EV(CAR FORLIS)),LEVEL,DUMMY)∧
        ALLFORFULINST(ACTION,(EV(CDR FORLIS)),LEVEL)

    →   ALLFORFULINST(ACTION,FORLIS,LEVEL)
```

Figure 2-22 is continued on the next page.

```
ALLFORDEF
        [NULL(FORLIS)∧DOACTION(ACTION,LEVEL)∧THSETQ(DUMMY,0)]∨
        [THSETQ(ITM1,(EV(CADAR FORLIS)))∧
            THSETQ(LEVEL1,(EV(CADDR(CAR FORLIS))))∧
            THSETQ(ITM2,(EV(CADR(CDDAR FORLIS))))∧
            THSETQ(LEVEL2,(EV(CADDR(CDDAR FORLIS))))∧
            THSETQ(REL,(EV(CAAR FORLIS)))∧
            DETVAL(ITM2,LEVEL2,ARG2,NIL,NIL)∧
            DETVAL(ITM1,LEVEL1,ARG1,ARG2,REL)∧
            TEST((EV(LIST REL ARG1 ARG2)),LEVEL)∧
            ALLFOR(ACTION,(EV(CDR FORLIS)),LEVEL,DUMMY)

    →   ALLFOR(ACTION,FORLIS,LEVEL,DUMMY)

DETVALDEF
        THSETQ(ARG,ITM)∧
        [LITRAL**(ITM)∨
            [=(ITM,RUNTIME)∧NEWVAR(ARG)∧GETRUNT(ARG,ARG2,REL)]∨
            [CALCKEY(REC,ITM)∧ISPORT(REC)∧=(REL,EQ)∧
                FOUND(REC,ITM,ARG2,LEVEL)∧
                EQ (ERRORSTATUS,0)]∨
            [ISITEM(ITM)∧CONTAINS(REC,ITM)∧
                INCORE(REC,LEVEL)]∨
            [[STAT**(ITM)∨REPQ**(ITM)∨RETQ**(ITM)]∧
                LINKS(ACTION,PREC,GREC,LEVEL)∧
                [¬BCA*(ITM)∨C((EV(CADR ACTION)),0)]∧
                [¬RETQ**(ITM)∨C((EV(CADR ACTION)),0)]∧
                [¬=(ITM,ONE)∨C((EV(CADR ACTION)),0)]∧
                [¬BTMMA*(ITM)∨[TOBEUSED(ITMS,LEVEL)∧
                    INITVARS**(ITMS)]]∧
                THSETQ(ARG,(EV(CADR ACTION)))∧
                LINKED(ACTION,PREC,GREC,LEVEL,ARG,ARG2)∧
                [¬=(ITM,AVE)∨
                    DIVVARS**((EV(CADR ACTION)),ITMS)]]]

    →   DETVAL(ITM,LEVEL,ARG,ARG2,REL)
```

Figure 2-22 is continued on the next page.

```
GETRUNTDEF
          EQ (VAR,HIGH-VALUES)∧ACCEPT(VAR,ITM,REL)

       → GETRUNT(VAR,ITM,REL)

DOACTIONDEF
          THSETQ(COMND,(EV(CAR ACTION)))∧
          THSETQ(ARG,(EV(CADR ACTION)))∧
          [¬REPQ**(COMND)∨
             [TOBEUSED(ITMS,LEVEL)∧DETALLVAL**(ITMS,NIL)∧
                BINDITML**(ITMS)∧
                [NULL(ITML)∨ANYOUTPUT(ITML,LEVEL)]∧
                NEXTLEVOUT**(ITMS)]∧
          [¬=(COMND,ALL)∨C(ARG,0)]∧
          [¬=(COMND,ANY)∨C(ARG,1)]∧
          [¬=(COMND,ONE)∨C(ARG,1)]∧
          [¬BCA*(COMND)∨C(ARG,(ADD1 ARG))]∧
          [¬BTMMA*(COMND)∨[TOBEUSED(ITMS,LEVEL)∧
             DETALLVAL**(ITMS,ACTION)]]

       → DOACTION(ACTION,LEVEL)

FREETOGOUP
          LINKS(DU1,DU2,REC,LEVEL)∧
          [[CURRENT(REC,LEVEL)∧UPLINKED(REC,GREC,LEVEL)]∨
             [CURRENT(XREC,LEVEL)∧UPLINKED(XREC,GREC,LEVEL)]]

       → CURRENT(GREC,LEVEL)

CURAX
          ¬NULL((EV(CAR LEVEL)))∧CURRENT(REC,(EV(CAR LEVEL)))

       → CURRENT(REC,LEVEL)

ITEMAX
          DBKEY(REC,ITM)∨CONTAINS(REC,ITM)

       → ISITEM(ITM)
```

Figure 2-22. Type S3 and S4 rules.

2.8.3 Type S3 and S4 rules.

In Figure 2-22 each rule is labeled, the pre-condition is described and then the post-condition is given preceeded by a "→". Four of these rules, ALLFORDEF, ALLFORFULINSTDEF, UPHIERARCHY, and CURAX are recursive.

2.8.3.1 The STAYUP rule.

The STAYUP rule checks for completion of an access path. The first disjunction in the pre-condition of STAYUP, consisting of the first four assertions, insures that either REC and GREC are the same and current or code has been generated linking REC to GREC with FIND OWNER operations only.

The next disjunction of two assertions forces the quantification flag for ALL (only if such quantification has been specified as the ACTION) to be temporarily false. If the condition test succeeds then the flag will be reset to true. At the end of the iteration, the flag will have its proper value.

The next disjunction of STAYUP (four assertions) specifies immediate construction of the code for the specified ACTION if the activity is not conditional. If the activity is conditional then ALLFORFULINST will insure generation of the code for conditional testing as well as actually performing the action. STAYUP is equivalent to the <ACTION> production.

Rather than describing all of the remaining rules in detail, Figure 2-23 gives

correspondences between the productions of Figure 2-11 and the rules. However, informal descriptions of several of the rules follow.

RULE	PRODUCTION
TOPPROG	<MAIN PARAGRAPH>
PORTDEF	<PORT>
STAYUP	<ACTION>
UPHIERARCHY	<FINDOWNER SEQUENCE> in <ACTION>
ALLFORFULINSTDEF	
ALLFORDEF	<CONDITION TEST>
DETVALDEF	<VALUE DETERMINATION>,<CONDITION VALUE>
GETRUNTDEF	<RUNTIME>
DOACTIONDEF	<DISPLAY SEQUENCE>,<STATISTIC SEQUENCE>
FREETOGOUP	<FINDOWNER SEQUENCE> in <VALUE DETERMINATION>

Figure 2-23. Rule and production correspondences.

2.8.3.2 The DOACTIONDEF rule.

DAOCTIONDEF contains many disjunctions of the form $\neg A \vee B$ which of course has the same effect as $A \supset B$.

ACTION is a list consisting of a command and a variable which is available for use in satisfying the command. The first assertion extracts the command from ACTION and stores it in COMND. The second assertion extracts the variable and stores it in VAR.

The first disjunction (incorporating the next seven assertions) specifies that if

the COMND is a report quantifier, then generate the following, in order:

(a) Code to determine the values for all ITMS to be used in this program segment (DETALLVAL##).

(b) Code to display the matrix (ANYOUTPUT).

(c) Code to handle all sub-levels within the context of the current level (NEXTLEVOUT##).

The next three disjunctions (six assertions) cause code to be generated to set ARG to zero or one if the COMND is a quantifier. The next to the last disjunction (three assertions) causes ARG to be incremented by one if the COMND is a COUNT or AVE.

The last disjunction in DOACTIONDEF causes code generation for processing the ITMS matrix according to the statistical command given in COMND. This code will either total all of the columns of the matrix or find the minimun or maximum entry in each column.

2.8.3.3 The ALLFORDEF rule.

ALLFORDEF is another important rule. This is a recursive rule which controls code generation for a conjunctive list of tests (contained in FORLIS). Each iteration of the rule will result in code generation for one such test. The first three assertions of ALLFORDEF will (a), terminate the recursion when all tests in FORLIS have been taken care of (FORLIS is empty) and (b), cause code generation for the conditional ACTION associated with FORLIS.

If FORLIS is not empty then the next five predicates, all THSETQ, extract the five parameters describing the first test in FORLIS. These five parameters were described in section 2.5.2. The next two assertions in ALLFORDEF, both DETVAL, insure code generation for the value determination of the two arguments which are to be compared in the test. The next assertion causes insertion of a conditional statement if the truth value of the relation is not known. The last assertion causes the recursion and insures the proper code generation for the remaining tests in FORLIS.

Section 2.5.2 stated that a complete condition is actually a disjunctive list of lists of test conjunctions, yet ALLFORDEF is apparently only processing a conjunctive list of tests. That is why ALLFORFULINSTDEF, another recursive rule, is included. This rule is entered once for each list of conjunctions, e.g. once for each disjunct in the complete condition. In other words, ALLFORFULINSTDEF passes the first list of tests (CAR FORLIS) to ALLFORDEF and then recursively enters itself with the remaining lists, e.g. (CDR FORLIS).

2.8.3.4 The CURAX axiom.

CURAX is a simple axiom. It is interesting because it is recursive and uses the Dewey-decimal structure of LEVEL. CURAX specifies that a record is CURRENT at a particular level if this level is within the context of another level in which the record is already CURRENT. In other words, to find out if a record is current at X.2.1, CURAX will see if it is current at X.2. (CAR LEVEL) returns the

value X.2 because X.2.1 is represented in a list structure as ((X.2).1).

2.8.3.5 The PORTDEF rule.

It was brought out earlier that there are three methods of entry to a DBTG data base. An entry point is called a port. The physical beginning of an area is a port, as is a record which can be directly accessed. Finally, DBTG lets the user define sets which are owned by the SYSTEM. What this really amounts to is that the data management system stores the direct access key of the first record of such a set in an internal table. A SYSTEM owned set is therefore directly accessible and can be used as a port. Since a set or area can contain more than one record, a program loop is required to access all of the records in the set or area.

PORTDEF contains three disjunctions in its precondition, one disjunction for each type of port. The type of port selected is preferred by PORTDEF in the following order: direct access, SYSTEM owned set, or Area search. The first two assertions will be true if a direct access is possible, and code has been constructed for this purpose. The second disjunct is true only if the top level record of the query (GREC) is directly or indirectly owned by the SYSTEM. If neither of the two preferred ports can be used then an Area search is forced by setting REC to the value AREA.

PORTDEF only generates code if a loop is used as a port. The code generation

for other ports occurs when the Program Writer attempts to satisfy the LINKED goal following the GETPORT goal in TOPPROG. Satisfying this goal will generate the code for the access path from PREC to GREC. Recall that GREC is the top-level context record and that PREC has been bound by GETPORT to one of three values: the name of the direct access record, the keyword SYSTEM or the keyword AREA. Note that even if a direct access port is used it need not be the top level record of the query.

2.9 Efficiency Considerations.

2.9.1 Code duplication.

The programs generated by the system are far from optimal. The generated programs are frequently larger than necessary because of the repetition of equivalent program blocks. This occurs if a particular block is needed in more than one location in the program.

2.9.2 Data characteristics.

Program construction also occurs in the absence of any information about data characteristics such as data volumes and distributions. As a result, arbitrary choices are frequently made where alternative program constructions are possible. One such arbitrary decision which can impact program efficiency depending on unknown (to the system) characteristics is port selection.

The system prefers to use a SYSTEM owned set as a port rather than an area search if a direct access port is not possible. Yet, if one of the target records occurs relatively infrequently in a (physically) small area, then the area search is preferred.

An area search may also be preferred if the alternative SYSTEM owned set "meanders" through the data base. Searching such a set can require more physical accesses than searching the entire area containing that set. This may occur if the unit of physical access, called a page, contains more than one

record. In Figure 2-24 each large box is a page of an area, and each "X" is a target record of the query. All of the target records are contained in a system owned set as illustrated by the arrows. It would require a total of seven page accesses to traverse the set, yet an area search would require only four accesses.

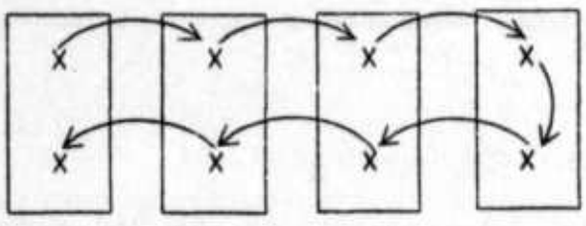


Figure 2-24. Area search vs a SYSTEM owned set search.

2.9.3 Conjunction testing.

Although, the system ignores efficiency factors in most of its decisions, there are some instances in which efficiency considerations are reflected in the programs constructed. One such instance is exhibited in the construction of tests for conjunctions. Rather than testing the entire conjunction in a single IF statement, each conjunct is tested separately. Although this may lead to a

somewhat larger program than necessary, the separation of conjuncts will lead to a more efficient execution.

The increased efficiency results because separating the tests minimizes the number of test preparations that will be necessary to evaluate the complete condition. Because all tests are part of a conjunction, the i-th test is only necessary if the preceeding (i - 1) tests are all true. Since a test preparation can involve many data base accesses (in the calculation of a statistic, for example), minimizing such test preparations can have a dramatic effect on program execution time and accesses to secondary storage.

2.9.4 Ordering of tests.

The system currently generates test evaluations in the order they are presented in the query. Re-ordering these tests according to cost of the necessary preparation (lowest first) and expected failure rate (highest first) would further increase execution efficiency.

2.9.5 Early termination of PERFORM.

Increased efficiency in program execution also results from the code generated by the system as described by the <SYSTEM TEST> production. The inclusion of a <SYSTEM TEST> in a <CONTROL TEST> is necessary only if the loop is controlled by the ONE quantifier. The <SYSTEM TEST> will terminate execution of the loop after one line of the matrix has been displayed. The use of

<SYSTEM TEST> with the logical quantifiers is not necessary but strongly suggested since there is no reason to continue execution of the loop as soon as existential (ANY) quantification has been proved or universal (ALL) quantification has been disproved. An example of this use can be seen in Figure 2-12.

A system test can also be used to terminate the execution of loops which calculate a COUNT, a MINima or a MAXima, if the result of the calculation is to be used in a test. This is possible because the value of such a calculation will vary monotonically during execution of a loop. The value of a COUNT or a MAX always increases. The value of a MIN will only get smaller. An AVErage will never behave monotonically unless values are first sorted, and a TOTal is monotonic only if its possible set of arguments contains numbers which are all of the same sign.

Therefore, if the value which is being calculated in the loop is to be compared with a known value, the loop can be terminated if the COUNT or MAX is greater (or the MIN is less) than the known value. Note that the actual relation used in the test is not used to determine the relation used in the system test. The effect of the system test can be made stronger in certain cases depending on the relation in which the statistic is to be tested. For example, if the test is "COUNT LT 10", then the system test can be the stronger "COUNT GE 10" rather than the standard "COUNT GT 10".

2.9 Efficiency Considerations.

DMLP generates the "standard" system test. The simple extension to generate stronger tests would definitely be warranted in a "production" implementation.

In determining the values of "A" and "B" in (A REL B), the system first determines the value of "B", then the value of "A". This permits early loop termination in determining the value of "A" because the value of "B" is known. Improving DMLP so that it can choose which of "A" or "B" to evaluate first would permit early loop termination when determining the value of "B" as well.

2.10 ALGOL TO COBOL Conversion.

Programs constructed by the APG system are written in an Algol-like language. The DMLP (which uses the APG) must translate this to COBOL and DML.

Algol [Naur et al 1960], as its name suggests, is an algorithmic language. The Algol-like language generated by the APG system uses labels very sparsely: program labels are only referenced at disjunctive branches in the program. At such a point of disjunction, the APG system generates a conditional branch to a subroutine which encompasses all alternatives but one. Then code generation continues for the excluded alternative. The current state and the goal for the sub-routines are saved. After completely generating the main program, the system recalls the goal and state, and commences program generation of the subroutines.

The conversion process from the Algol-like language transforms each such subroutine to a sub-section of the COBOL program. The main routine becomes the main section of the program. This explains the distinction in the BNF description of section 2.6 between <MAIN SECTION> and <SUB SECTION>. The conditional branch in the COBOL program is effected with a PERFORM of the first paragraph in the sub-section. The first paragraph in the section is performed (rather than the section itself) to insure that program control will not flow sequentially out of the first paragraph.

The need for sub-sections only arises if the query contains a disjunctive condition. The sub-section is entered only if the first disjunct fails. The first step in the sub-section tests the second disjunct. DMLP generates a branch to a second sub-section if there is a third disjunct in the condition. If there are no further disjuncts in the condition, then DMLP inserts the clause "NEXT SENTENCE" in place of the conditional branch.

Labels are used in the Algol-like programs only to transfer program control to sub-routines. All program control within a routine is accomplished via appropriate nesting of program blocks. Although COBOL allows nesting of IF's, the nested IF's give a program a messy appearance, much less clear, in fact, than the equivalent in Algol. Furthermore, COBOL does not allow the nesting of blocks which may be repetitively executed. Each such block must be either a named paragraph, a named section or a contiguous set of named paragraphs or named sections. That is not to say that program control cannot be logically nested; e.g., it is perfectly legal in COBOL to repetitively execute a block from a block which is being repetitively executed. The point is that in COBOL the nesting is not physical in the program text; instead, it is accomplished by labeling the blocks and then using the labels as the object of a PERFORM.

Transformation from Algol structure to COBOL structure requires changing a nested block structure to a sequentially labeled block structure. The process of this transformation is simple.

The complete Algol-like program is stored in a LISP list structure in the obvious way; e.g., each nested segment forms a sub-list. Conversion proceeds as follows; translate each top level element of the list to the appropriate program statement unless the element is itself a list, e.g., a nested program block. If such a nested block is encountered then generate a new paragraph name and a program statement which performs the new name, extract the sub-list containing the nested block from the main list and push the new paragraph name followed by each element of the sub-list onto the end of the main list. Further levels of nesting are then properly processed in due course because each such block eventually becomes a top level element of the main list.

In addition to structural differences between Algol and COBOL, there are also syntactic differences. The syntactic conversion is accomplished by a simple table look up process in a table of clause "masks". This process also cleans up such things as double negatives in logical statements. For example, -LE is changed in two steps, first to GT and finally to "IS GREATER THAN".

Figure 2-25, gives the Algol-like program generated by the APG system for the query of Figure 2-6. Contrast this with Figure 2-26 which is the COBOL equivalent of Figure 2-25.

The generated COBOL would be completely consistent with the specification in [CODASYL 1970] if certain instances of the MOVE statement are replaced by a

COMPUTE statement. The generated programs may also make use of a MIN or MAX function which is not standard in COBOL.

THE:GOAL: (PROGRAM P1):IS:ATTAINABLE:BY:THE:FOLLOWING:PROGRAM:

```
PROC1 (P1)
  BEGIN <-------------------------------------------------------------- PARA100
  OPEN((A1 A2));
  FINDFIRST(AREA DOCTOR A1);
  DDBKEY ← CURSTAT(A1 AREA);
  WHILE ¬NE (ERRORSTATUS 0) DO
    BEGIN <---------------------------------------------------- PARA101
    Z2 ← CURSTAT(A1 AREA);
    GET(DOCTOR);
    DISPLAY((DOCNAME DOCAGE SPECIALTY));
    FINDFIRST(SET TREATMENT TREATING);
    TDBKEY ← CURSTAT(TREATING SET);
    WHILE ¬NE (ERRORSTATUS 0) DO
      BEGIN <-------------------------------------------- PARA102
      Z1 ← CURSTAT(TREATING SET);
      FINDOWNER(TREATMENTS);
      GET(PATIENT);
      IF ¬TEST((GT PATAGE (21)) (X . 1)) THEN
        PROC2
      ELSE
        BEGIN <---------------------------- 103
        GET(TREATMENT);
        DISPLAY((PATNAME PATAGE DIAGNOSIS));
        END <-----------------------------
      TDBKEY ← Z1
      FINDUSING(TREATMENT TDBKEY)
      FINDNEXT(SET TREATMENT TREATING)
      END <-----------------------------------------------
    DDBKEY ← Z2;
    FINDUSING(DOCTOR DDBKEY);
    FINDNEXT(AREA DOCTOR A1);
    END <-------------------------------------------------------
  CLOSE((A1 A2));
  STOP(P1);
  END <-------------------------------------------------------------------
```

Figure 2-25. Program P1 (Figure 2-6) in Algol.

THE:GOAL: (PROGRAM P1):IS:ATTAINABLE:BY:THE:FOLLOWING:PROGRAM:

```
PROCEDURE DIVISION.
PROC1 SECTION.
PARA-100.
   OPEN AREA A1 A2 .
   FIND FIRST DOCTOR RECORD OF A1 AREA .
   MOVE CURRENCY STATUS FOR A1 AREA TO DDBKEY .
   PERFORM PARA-101 UNTIL ERRORSTATUS IS NOT EQUAL TO 0 .
   CLOSE AREA A1 A2 .
   STOP .
PARA-101.
   MOVE CURRENCY STATUS FOR A1 AREA TO Z2 .
   GET DOCTOR RECORD .
   DISPLAY DOCNAME DOCAGE SPECIALTY .
   FIND FIRST TREATMENT RECORD OF TREATING SET .
   MOVE CURRENCY STATUS FOR TREATING SET TO TDBKEY .
   PERFORM PARA-102 UNTIL ERRORSTATUS IS NOT EQUAL TO 0 .
   MOVE Z2 TO DDBKEY .
   FIND DOCTOR USING DDBKEY .
   FIND NEXT DOCTOR RECORD OF A1 AREA .
PARA-102.
   MOVE CURRENCY STATUS FOR TREATING SET TO Z1 .
   FIND OWNER RECORD OF TREATMENTS SET .
   GET PATIENT RECORD .
   IF PATAGE IS NOT GREATER THAN 21  NEXT SENTENCE
      ELSE PERFORM PARA-103.
   MOVE Z1 TO TDBKEY .
   FIND TREATMENT USING TDBKEY .
   FIND NEXT TREATMENT RECORD OF TREATING SET .
PARA-103.
   GET TREATMENT RECORD .
   DISPLAY PATNAME PATAGE DIAGNOSIS .
```

Figure 2-26. Program P1 in COBOL.

2.11 Examples of Procedure Generation.

This section concludes Chapter 2 with several examples of procedure generation. For each example the query is given followed by the generated COBOL program. The specific features demonstrated by each query are classified and indexed in Figure 2-27. Program P1 of the feature index is the program of Figure 2-26.

Two different data base structures were used for these examples. The first is for a community medical data base. This data base contains information on all hospitals, doctors and patients in a community. The structure for this data base is illustrated in Figure 2-28. The data structure diagrams use the following conventions: Record, sel, and area names are in upper case, item names are in lower case, and calculated keys are underlined. Area boundaries are indicated with a dashed line. Other conventions of DS diagrams were discussed in Chapter One.

Each record type also has a working storage register associated with it which the DBTG calls a data base key. This data base key can be used to store the data base address of a record occurence. The data base key for each record has been indicated in parenthesis.

2.11 Examples of Procedure Generation.

PROGRAM:	P1	P2	P3	P4	P5	P6
CONDITIONS:						
COMPLEX		X	X	X		
DISJUNCTION		X				
CONJUNCTION			X	X		
QUANTIFICATION			X			
USE OF STATISTIC		X		X		
DISPLAY						
NESTED MATRICES	X	X	X		X	X!
CONFLUENT HIERARCHY	X			X		
DOWNWARD MIGRATION			X	X	X!	
STATISTIC			X			
CONDITIONAL				X	X!	
"ONE" COMMAND				X		
INTERACTION:						
CALCKEY VALUE		X			X	X
OTHER VALUE			X	X		
PORT:						
CALCKEY		X			X	X
SYSTEM SET			X	X		
AREA SEARCH	X					
DATA BASE:						
MEDICAL	X	X				X
SALES			X	X	X	

Figure 2-27. Index to the examples.

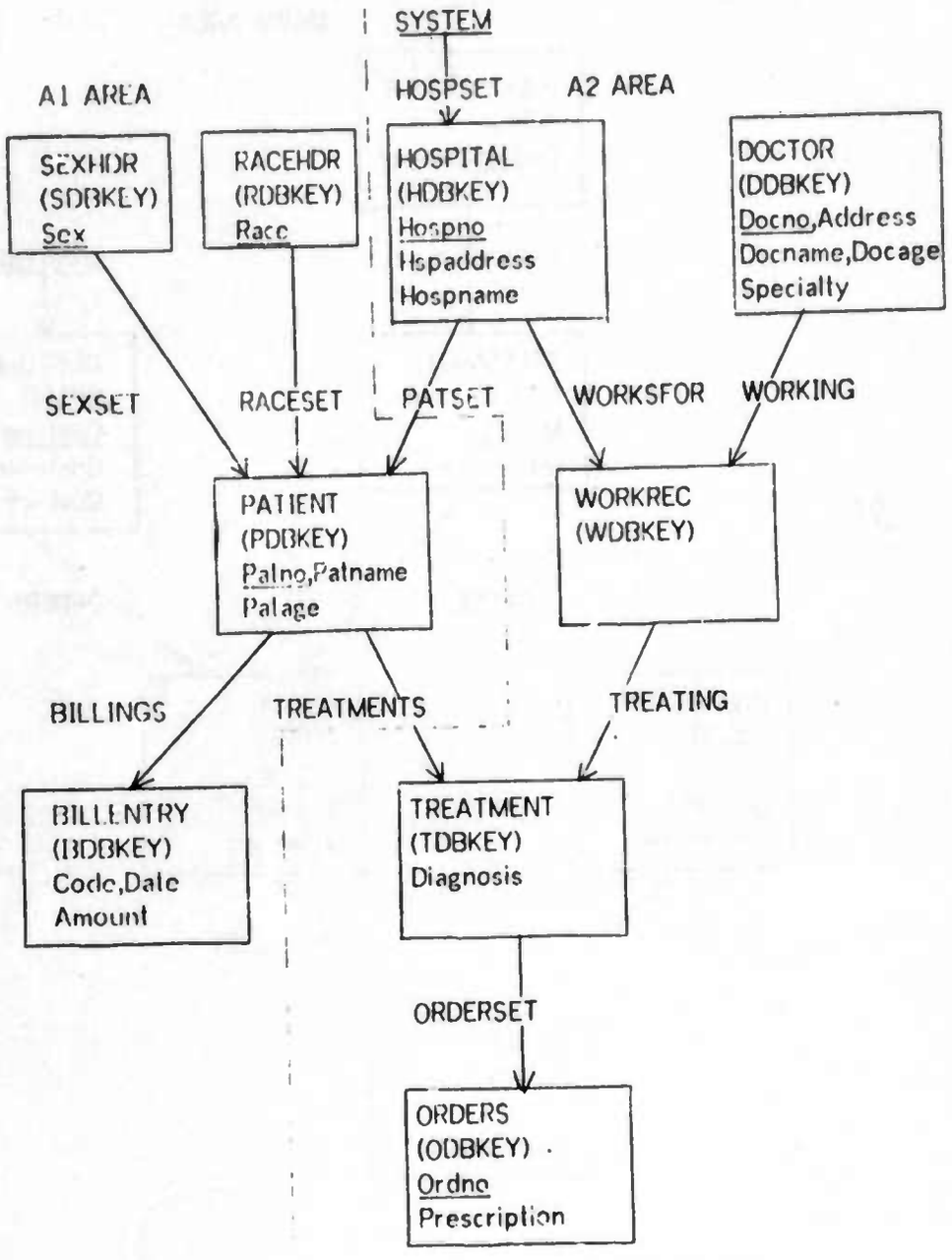


Figure 2-28. A community medical data base structure.

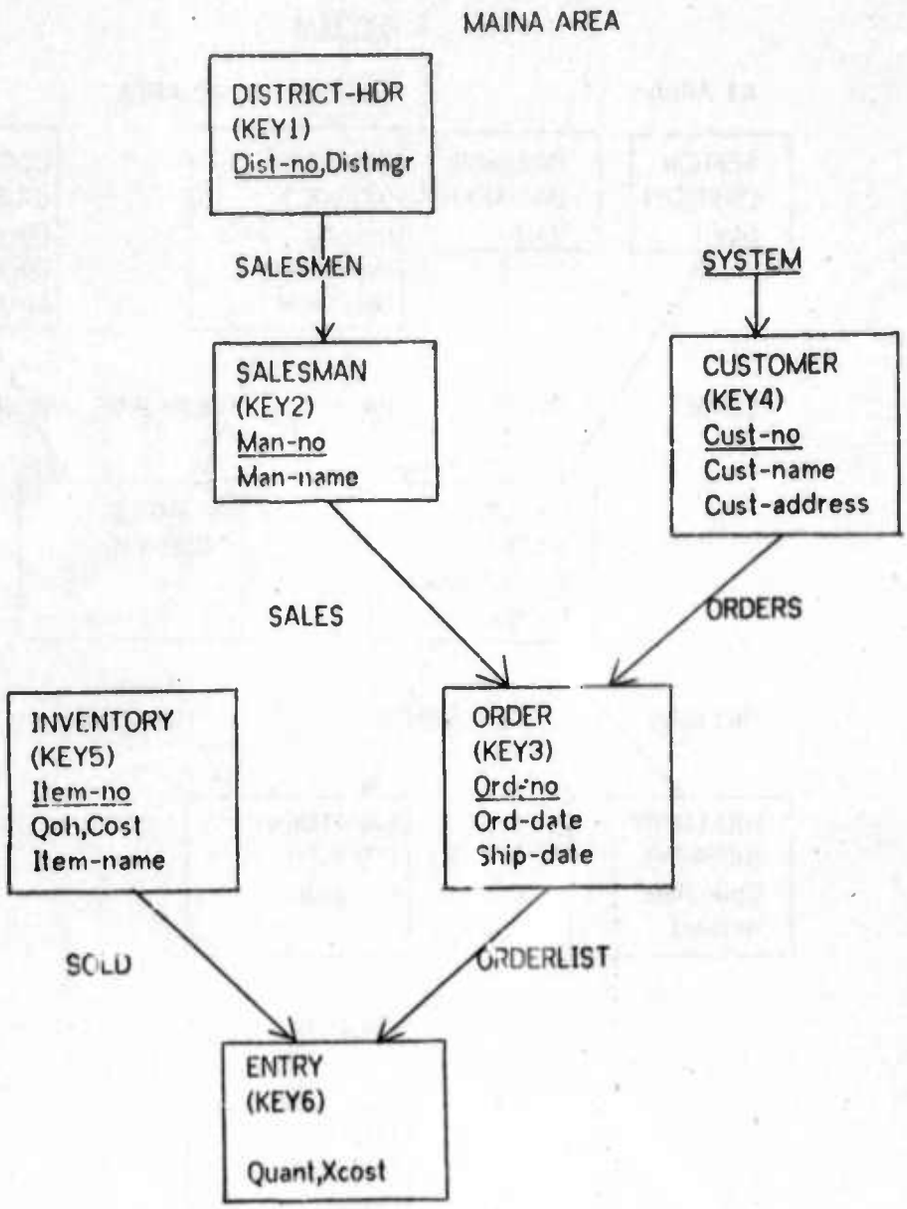


Figure 2-29. A sales data base structure.

Figure 2-29 illustrates the other data base structure used in the examples. This data base might be the sales portion of an inventory system.

Program P2 (Figures 2-30, 2-31) will list (for a hospital specified at execution time) all patients who have accumulated a total uninsured billing of over $200. Uninsured billings are identified with one of two billing codes. This disjunction leads to the separate section called PROC3. P2 illustrates the unnecessary generation of duplicate paragraphs that may occur: PARA-105 is identical to PARA-301.

Program P3 (Figures 2-32, 2-33) lists the average dollar value and average number of unique items per order received for all active salesmen. The set of active salesmen is defined at run time by giving the time interval during which sales activity must have occurred. P3 makes extensive use of system generated variables. X1 is the existential quantifier flag; X13 and X14 contain values entered by the user at run-time. X4 accumulates a count of the number of orders per salesman. X7 contains the total sold by a salesman. X6 contains the total value of an order. X10 contains the number of entries per order. X11 contains the number of entries per salesman. X7 and X11 are reset to contain averages in the last three statements of PARA-103.

Program P4 (Figures 2-34, 2-35) identifies possible unhappy customers (more than three unfilled orders received prior to an interactively specified date) and especially flags "important customers", those whose recent orders exceed $10,000.

Program P5 (Figures 2-36, 2-37) simply lists any specified order and indicates which items, if any, cannot be filled from inventory. This program makes heavy use of downward migration, especially in PARA-101.

Program P6 (Figures 2-38, 2-39) is another one using the medical data base. It generates a doctor's cross reference: for a particular doctor, each patient and all of each patient's doctors are listed. It is for programs like these, which traverse a confluent hierarchy in two directions, that conservation of the loop invariant becomes important. The loop invariant is the current of set.

The report generated by P6 will list the top level doctor in many locations: at the beginning of the report and with each patient (because he is one of the doctors associated with each of his patients). This is a minor deficiency but it can be cured with a simple extension to the DMLP. Providing for the definition of temporary variables would allow the user to differentiate between doctors at different levels of the query. This would be accomplished by specifying the storage of the doctor's number in a temporary variable at the top level of the query; e.g. SAVE DOCNO IN TDOC. Then retrieval at the third level would be specified to be conditional on (DOCNO NE TDOC).

```
ENTER PROGRAM NAME  P2
READ DSK:?  T
  PRIMARY RECORD (MAIN)
  *HOSPITAL
  CONDITIONS FOR RETRIEVAL
  *(HOSPNO EQ RUNTIME)
  *NIL
  ITEMS OR STATS TO BE DISPLAYED
  *HOSPNAME
  *HOSPNO
  *REPEAT
    PRIMARY RECORD (REPEAT)
    *PATIENT
    CONDITIONS FOR RETRIEVAL
    *(TOT GE 200)
      PRIMARY RECORD (TOT)
      *BILLENTRY
      CONDITIONS FOR RETRIEVAL
      *(CODE EQ 'X')
      *OR
      *(CODE EQ 'Z')
      *NIL
      ITEMS OR STATS FOR TOT
      *AMOUNT
      *NIL
    *NIL
    ITEMS OR STATS TO BE DISPLAYED
    *PATNAME
    *PATNO
    *NIL
  *NIL
POSSIBLE PORTS ARE:
(HOSPNO)
SELECT ONE OR TYPE NIL  HOSPNO
```

Figure 2-30. Query P2: "For a hospital specified at run-time, list its name and number and the name and number of all patients whose total billings of code 'X' or 'Z' exceeds or equals $200."

2.11 Examples of Procedure Generation.

THE:GOAL: (PROGRAM P2):IS:ATTAINABLE:BY:THE:FOLLOWING:PROGRAM:

```
PROCEDURE DIVISION.
PROC1 SECTION.

PARA-100.
   OPEN AREA A1 A2 .
   DISPLAY 'HOSPNO' 'EOJ' .
   ACCEPT HOSPNO .
   FIND HOSPITAL RECORD .
   IF ERRORSTATUS IS NOT EQUAL TO 0  NEXT SENTENCE
      ELSE PERFORM PARA-101.
   CLOSE AREA A1 A2 .
   STOP .

PARA-101.
   GET HOSPITAL RECORD .
   DISPLAY HOSPNAME HOSPNO .
   FIND FIRST PATIENT RECORD OF PATSET SET .
   MOVE CURRENCY STATUS FOR PATSET SET TO PDBKEY .
   PERFORM PARA-102 UNTIL ERRORSTATUS IS NOT EQUAL TO 0 .

PARA-102.
   MOVE CURRENCY STATUS FOR PATSET SET TO Z2 .
   MOVE 0 TO X5 .
   FIND FIRST BILLENTRY RECORD OF BILLINGS SET .
   MOVE CURRENCY STATUS FOR BILLINGS SET TO BDBKEY .
   PERFORM PARA-103 UNTIL ERRORSTATUS IS NOT EQUAL TO 0 .
   IF X5 IS LESS THAN 200  NEXT SENTENCE
      ELSE PERFORM PARA-104.
   MOVE Z2 TO PDBKEY .
   FIND PATIENT USING PDBKEY .
   FIND NEXT PATIENT RECORD OF PATSET SET .

PARA-103.
   MOVE CURRENCY STATUS FOR BILLINGS SET TO Z1 .
   GET BILLENTRY RECORD .
   IF CODE IS NOT EQUAL TO 'X'  PERFORM PARA-300
      ELSE PERFORM PARA-105.
   MOVE Z1 TO BDBKEY .
   FIND BILLENTRY USING BDBKEY .
   FIND NEXT BILLENTRY RECORD OF BILLINGS SET .

PARA-105.
   MOVE X5 + AMOUNT TO X5 .
```

```
PARA-104.
   GET PATIENT RECORD .
   DISPLAY PATNAME PATNO .

PROC3 SECTION.
PARA-300.
   IF CODE IS NOT EQUAL TO 'Z' NEXT SENTENCE
      ELSE PERFORM PARA-301.

PARA-301.
   MOVE X5 + AMOUNT TO X5 .
```

Figure 2-31. Program P2.

```
ENTER PROGRAM NAME  P3
READ DSK:?  T
  PRIMARY RECORD (MAIN)
  *SALESMAN
  CONDITIONS FOR RETRIEVAL
  *ANY
    PRIMARY RECORD (ANY)
    *ORDER
    CONDITIONS FOR RETRIEVAL
    *(ORD-DATE GE RUNTIME)
    *(ORD-DATE LE RUNTIME)
    *NIL
  *NIL
  ITEMS OR STATS TO BE DISPLAYED
  *MAN-NAME
  *DIST-NO
  *AVE
    PRIMARY RECORD (AVE)
    *ORDER
    CONDITIONS FOR RETRIEVAL
    *NIL
    ITEMS OR STATS FOR AVE
    *TOT
      PRIMARY RECORD (TOT)
      *ENTRY
      CONDITIONS FOR RETRIEVAL
      *NIL
      ITEMS OR STATS FOR TOT
      *XCOST
      *NIL
    *COUNT
      PRIMARY RECORD (COUNT)
      *ENTRY
      CONDITIONS FOR RETRIEVAL
      *NIL
    *NIL
  *NIL
```

Figure 2-32. Query P3: "For those salesmen who have any orders with an order date that falls between two dates specified at run-time, display his name, district number, the average cost per order, and the average number of items per order."

THE:GOAL: (PROGRAM P3):IS:ATTAINABLE:BY:THE:FOLLOWING:PROGRAM:

```
PROCEDURE DIVISION.
PROC1 SECTION.
PARA-100.
   OPEN AREA MAINA .
   FIND FIRST SALESMAN RECORD OF MAINA AREA .
   MOVE CURRENCY STATUS FOR MAINA AREA TO KEY2 .
   PERFORM PARA-101 UNTIL ERRORSTATUS IS NOT EQUAL TO 0 .
   CLOSE AREA MAINA .
   STOP .
PARA-101.
   MOVE CURRENCY STATUS FOR MAINA AREA TO Z5 .
   MOVE 0 TO X1 .
   FIND FIRST ORDER RECORD OF SALES SET .
   MOVE CURRENCY STATUS FOR SALES SET TO KEY3 .
   PERFORM PARA-102 UNTIL
      ( ERRORSTATUS IS NOT EQUAL TO 0 ) OR
      ( X1 IS EQUAL TO 1 ) .
   IF X1 IS NOT EQUAL TO 1  NEXT SENTENCE
      ELSE PERFORM PARA-103.
   MOVE Z5 TO KEY2 .
   FIND SALESMAN USING KEY2 .
   FIND NEXT SALESMAN RECORD OF MAINA AREA .
PARA-102.
   MOVE CURRENCY STATUS FOR SALES SET TO Z1 .
   IF X13 IS NOT EQUAL TO HIGH-VALUES  NEXT SENTENCE
      ELSE PERFORM PARA-104.
   GET ORDER RECORD .
   IF ORD-DATE IS LESS THAN X13  NEXT SENTENCE
      ELSE PERFORM PARA-105.
   MOVE Z1 TO KEY3 .
   FIND ORDER USING KEY3 .
   FIND NEXT ORDER RECORD OF SALES SET .
PARA-104.
   DISPLAY 'ORD-DATE' 'GE?' .
   ACCEPT X13 .
PARA-105.
   IF X14 IS NOT EQUAL TO HIGH-VALUES  NEXT SENTENCE
      ELSE PERFORM PARA-106.
   IF ORD-DATE IS GREATER THAN X14  NEXT SENTENCE
      ELSE PERFORM PARA-107.
PARA-106.
   DISPLAY 'ORD-DATE' 'LE?' .
   ACCEPT X14 .
PARA-107.
```

```
    MOVE 1 TO X1 .
PARA-103.
    GET SALESMAN RECORD .
    FIND OWNER RECORD OF SALESMEN SET .
    GET DISTRICT-HDR RECORD .
    MOVE 0 TO X4 .
    MOVE 0 TO X7 .
    MOVE 0 TO X11 .
    FIND FIRST ORDER RECORD OF SALES SET .
    MOVE CURRENCY STATUS FOR SALES SET TO KEY3 .
    PERFORM PARA-110 UNTIL ERRORSTATUS IS NOT EQUAL TO 0 .
    MOVE X7 DIVIDED BY X4 TO X7 .
    MOVE X11 DIVIDED BY X4 TO X11 .
    DISPLAY MAN-NAME DIST-NO X7 X11 .
PARA-110.
    MOVE CURRENCY STATUS FOR SALES SET TO Z4 .
    MOVE X4 + 1 TO X4 .
    MOVE 0 TO X6 .
    FIND FIRST ENTRY RECORD OF ORDERLIST SET .
    MOVE CURRENCY STATUS FOR ORDERLIST SET TO KEY6 .
    PERFORM PARA-111 UNTIL ERRORSTATUS IS NOT EQUAL TO 0 .
    MOVE X7 + X6 TO X7 .
    MOVE 0 TO X10 .
    FIND FIRST ENTRY RECORD OF ORDERLIST SET .
    MOVE CURRENCY STATUS FOR ORDERLIST SET TO KEY6 .
    PERFORM PARA-112 UNTIL ERRORSTATUS IS NOT EQUAL TO 0 .
    MOVE X11 + X10 TO X11 .
    MOVE Z4 TO KEY3 .
    FIND ORDER USING KEY3 .
    FIND NEXT ORDER RECORD OF SALES SET .
PARA-111.
    MOVE CURRENCY STATUS FOR ORDERLIST SET TO Z2 .
    GET ENTRY RECORD .
    MOVE X6 + XCOST TO X6 .
    MOVE Z2 TO KEY6 .
    FIND ENTRY USING KEY6 .
    FIND NEXT ENTRY RECORD OF ORDERLIST SET .
PARA-112.
    MOVE CURRENCY STATUS FOR ORDERLIST SET TO Z3 .
    MOVE X10 + 1 TO X10 .
    MOVE Z3 TO KEY6 .
    FIND ENTRY USING KEY6 .
    FIND NEXT ENTRY RECORD OF ORDERLIST SET .
```

Figure 2-33. Program P3.

THE:GOAL: (PROGRAM P4):IS:ATTAINABLE:BY:THE:FOLLOWING:PROGRAM:

```
       PROCEDURE DIVISION.
       PROC1 SECTION.
       PARA-100.
          OPEN AREA MAINA .
          FIND FIRST CUSTOMER RECORD OF CUSTOMERS SET .
          MOVE CURRENCY STATUS FOR CUSTOMERS SET TO KEY4 .
          PERFORM PARA-101 UNTIL ERRORSTATUS IS NOT EQUAL TO 0 .
          CLOSE AREA MAINA .
          STOP .
       PARA-101.
          MOVE CURRENCY STATUS FOR CUSTOMERS SET TO Z5 .
          MOVE 0 TO X1 .
          FIND FIRST ORDER RECORD OF ORDERS SET .
          MOVE CURRENCY STATUS FOR ORDERS SET TO KEY3 .
          PERFORM PARA-10[illegible] UNTIL
             ( ERRORSTATUS IS NOT EQUAL TO 0 ) OR
             ( X1 IS GREATER THAN 3 ) .
          IF X1 IS NOT GREATER THAN 3  NEXT SENTENCE
             ELSE PERFORM PARA-103.
          MOVE Z5 TO KEY4 .
          FIND CUSTOMER USING KEY4 .
          FIND NEXT CUSTOMER RECORD OF CUSTOMERS SET .
       PARA-102.
          MOVE CURRENCY STATUS FOR ORDERS SET TO Z1 .
          IF X15 IS NOT EQUAL TO HIGH-VALUES  NEXT SENTENCE
             ELSE PERFORM PARA-104.
          GET ORDER RECORD .
          IF ORD-DATE IS NOT LESS THAN X15  NEXT SENTENCE
             ELSE PERFORM PARA-105.
          MOVE Z1 TO KEY3 .
          FIND ORDER USING KEY3 .
          FIND NEXT ORDER RECORD OF ORDERS SET .
       PARA-104.
          DISPLAY 'ORD DATE' 'LT?' .
          ACCEPT X15 .
       PARA-105.
          IF SHIP-DATE IS NOT EQUAL TO 0  NEXT SENTENCE
             ELSE PERFORM PARA-106.
       PARA-106.
          MOVE X1 + 1 TO X1 .
       PARA-103.
          GET CUSTOMER RECORD .
          DISPLAY CUST-NAME .
          MOVE 0 TO X3 . .
```

```
      FIND FIRST ORDER RECORD OF ORDERS SET .
      MOVE CURRENCY STATUS FOR ORDERS SET TO KEY3 .
      PERFORM PARA-107 UNTIL ( ERRORSTATUS IS NOT EQUAL TO 0 ) OR
         ( X3 IS EQUAL TO 1 ) .
      MOVE 0 TO X11 .
      FIND FIRST ORDER RECORD OF ORDERS SET .
      MOVE CURRENCY STATUS FOR ORDERS SET TO KEY3 .
      PERFORM PARA-110 UNTIL ERRORSTATUS IS NOT EQUAL TO 0 .
      IF X11 IS NOT GREATER THAN 10000  NEXT SENTENCE
         ELSE PERFORM PARA-111.
   PARA-107.
      MOVE CURRENCY STATUS FOR ORDERS SET TO Z2 .
      FIND OWNER RECORD OF SALES SET .
      FIND OWNER RECORD OF SALESMEN SET .
      GET DISTRICT-HDR RECORD .
      GET SALESMAN RECORD .
      DISPLAY DIST-NO MAN-NAME .
      MOVE 1 TO X3 .
      MOVE Z2 TO KEY3 .
      FIND ORDER USING KEY3 .
      FIND NEXT ORDER RECORD OF ORDERS SET .
   PARA-110.
      MOVE CURRENCY STATUS FOR ORDERS SET TO Z4 .
      FIND FIRST ENTRY RECORD OF ORDERLIST SET .
      MOVE CURRENCY STATUS FOR ORDERLIST SET TO KEY6 .
      PERFORM PARA-112 UNTIL ERRORSTATUS IS NOT EQUAL TO 0 .
      MOVE Z4 TO KEY3 .
      FIND ORDER USING KEY3 .
      FIND NEXT ORDER RECORD OF ORDERS SET .
   PARA-112.
      MOVE CURRENCY STATUS FOR ORDERLIST SET TO Z3 .
      GET ENTRY RECORD .
      MOVE X11 + XCOST TO X11 .
      MOVE Z3 TO KEY6 .
      FIND ENTRY USING KEY6 .
      FIND NEXT ENTRY RECORD OF ORDERLIST SET .
   PARA-111.
      DISPLAY 'IMPORTANT' 'CUSTOMER!' .
```

Figure 2-35. Program P4.

```
ENTER PROGRAM NAME  P5
READ DSK:?  T
  PRIMARY RECORD (MAIN)
  *ORDER
  CONDITIONS FOR RETRIEVAL
  *(ORD-NO EQ RUNTIME)
  *NIL
  ITEMS OR STATS TO BE DISPLAYED
  *ORD-NO
  *ORD-DATE
  *DIST-NO
  *MAN-NAME
  *CUST-NAME
  *CUST-ADDRESS
  *REPEAT
    PRIMARY RECORD (REPEAT)
    *ENTRY
    CONDITIONS FOR RETRIEVAL
    *NIL
    ITEMS OR STATS TO BE DISPLAYED
    *ITEM-NAME
    *QUANT
    *COST
    *XCOST
    *COND
      CONDITIONS FOR RETRIEVAL
      *(QUANT GT QOH)
      *NIL
      ITEMS OR STATS TO BE DISPLAYED
      *'STOCKOUT!'
      *'ONLY'
      *QOH
      *'AVAILABLE'
      *NIL
    *NIL
  *NIL
POSSIBLE PORTS ARE:
(ORD-NO)
SELECT ONE OR TYPE NIL  ORD-NO
```

Figure 2-36. Query P5: "For the run-time order, display the order number, date, district number salesman name, customer name, and customer address. Also display the name, quantity ordered, cost and extended cost of every item on the order. If the order cannot be filled, display the amount actually available in the message 'stockout! only -- available'."

THE:GOAL: (PROGRAM P5):IS:ATTAINABLE.3Y:THE:FOLLOWING:PROGRAM:

```
PROCEDURE DIVISION.
PROC1 SECTION.
PARA-100.
   OPEN AREA MAINA .
   DISPLAY 'ORD-NO' 'EQ?' .
   ACCEPT ORD-NO .
   FIND ORDER RECORD .
   IF ERRORSTATUS IS NOT EQUAL TO 0  NEXT SENTENCE
      ELSE PERFORM PARA-101.
   CLOSE AREA MAINA .
   STOP .
PARA-101.
   GET ORDER RECORD .
   FIND OWNER RECORD OF SALES SET .
   FIND OWNER RECORD OF SALESMEN SET .
   GET DISTRICT-HDR RECORD .
   GET SALESMAN RECORD .
   FIND OWNER RECORD OF ORDERS SET .
   GET CUSTOMER RECORD .
   DISPLAY ORD-NO ORD-DATE DIST-NO MAN-NAME CUST-NAME
      CUST-ADDRESS .
   FIND FIRST ENTRY RECORD OF ORDERLIST SET .
   MOVE CURRENCY STATUS FOR ORDERLIST SET TO KEY6 .
   PERFORM PARA-102 UNTIL ERRORSTATUS IS NOT EQUAL TO 0 .
PARA-102.
   MOVE CURRENCY STATUS FOR ORDERLIST SET TO Z1 .
   FIND OWNER RECORD OF SOLD SET .
   GET INVENTORY RECORD .
   GET ENTRY RECORD .
   DISPLAY ITEM-NAME QUANT COST XCOST .
   GET INVENTORY RECORD .
   GET ENTRY RECORD .
   IF QUANT IS NOT GREATER THAN QOH  NEXT SENTENCE
      ELSE PERFORM PARA-103.
   MOVE Z1 TO KEY6 .
   FIND ENTRY USING KEY6 .
   FIND NEXT ENTRY RECORD OF ORDERLIST SET .
PARA-103.
   DISPLAY 'STOCKOUT!' 'ONLY' QOH 'AVAILABLE' .
```

Figure 2-37. Program P5.

```
ENTER PROGRAM NAME  P6
READ DSK:?  T
  PRIMARY RECORD (MAIN)
  *DOCTOR
  CONDITIONS FOR RETRIEVAL
  *(DOCNO EQ RUNTIME)
  *NIL
  ITEMS OR STATS TO BE DISPLAYED
  *DOCNO
  *DOCNAME
  *REPEAT

    PRIMARY RECORD (REPEAT)
    *PATIENT
    CONDITIONS FOR RETRIEVAL
    *NIL
    ITEMS OR STATS TO BE DISPLAYED
    *PATNO
    *PATNAME
    *REPEAT

      PRIMARY RECORD (REPEAT)
      *DOCTOR
      CONDITIONS FOR RETRIEVAL
      *NIL
      ITEMS OR STATS TO BE DISPLAYED
      *DOCNO
      *DOCNAME
      *NIL
    *NIL
  *NIL
POSSIBLE PORTS ARE:
(DOCNO)
SELECT ONE OR TYPE NIL  DOCNO
```

Figure 2-38. Query P6: "Display the name and number of a doctor specified at run-time. Also display the name and number of all of his patients, and for each patient display the name and number of all of his doctors."

THE:GOAL: (PROGRAM P6):IS:ATTAINABLE:BY:THE:FOLLOWING:PROGRAM:

```
PROCEDURE DIVISION.
PROC1 SECTION.

PARA-100.
   OPEN AREA A1 A2 .
   DISPLAY 'DOCNO' 'EQ?' .
   ACCEPT DOCNO .
   FIND DOCTOR RECORD .
   IF ERRORSTATUS IS NOT EQUAL TO 0  NEXT SENTENCE
      ELSE PERFORM PARA-101.
   CLOSE AREA A1 A2 .
   STOP .

PARA-101.
   GET DOCTOR RECORD .
   DISPLAY DOCNO DOCNAME .
   FIND FIRST TREATMENT RECORD OF TREATING SET .
   MOVE CURRENCY STATUS FOR TREATING SET TO TDBKEY .
   PERFORM PARA-102 UNTIL ERRORSTATUS IS NOT EQUAL TO 0 .

PARA-102.
   MOVE CURRENCY STATUS FOR TREATING SET TO Z2 .
   FIND OWNER RECORD OF TREATMENTS SET .
   GET PATIENT RECORD .
   DISPLAY PATNO PATNAME .
   FIND FIRST TREATMENT RECORD OF TREATMENTS SET .
   MOVE CURRENCY STATUS FOR TREATMENTS SET TO TDBKEY .
   PERFORM PARA-103 UNTIL ERRORSTATUS IS NOT EQUAL TO 0 .
   MOVE Z2 TO TDBKEY .
   FIND TREATMENT USING TDBKEY .
   FIND NEXT TREATMENT RECORD OF TREATING SET .

PARA-103.
   MOVE CURRENCY STATUS FOR TREATMENTS SET TO Z1 .
   FIND OWNER RECORD OF TREATING SET .
   GET DOCTOR RECORD .
   DISPLAY DOCNO DOCNAME .
   MOVE Z1 TO TDBKEY .
   FIND TREATMENT USING TDBKEY .
   FIND NEXT TREATMENT RECORD OF TREATMENTS SET .
```

Figure 2-39. Program P6.

3 DESIGN OF DATA STRUCTURES.

3.1 Introduction.

3.1.1 Functional design.

The Data Base Task Group (DBTG) [CODASYL 1971a] describes a new job function called Data Base Administrator(DBA):

> "He will:
> Employ a data structure(s) that models the business or problem... Assign names in such a manner as to assure their uniqueness. Select search strategies..."

In other words, the DBA designs data structures. Although a structure can be defined existentially, as suggested by DBTG, "...models the business...", such a data base may contain information which will never be used. In addition, the quality of such a data base is heavily dependent on the DBA's ability to recognize the environment that must be captured (modeled) in the data base. It may be much better to define a structure functionally, e.g. based on anticipated information demands that the data base must satisfy.

In this chapter I will first describe how the DBA can utilize the retrieval program generator from the previous chapter as an aid in functional data base design. Following this, I will describe an automatic data base structure designer, hereafter called the designer, automatic designer, or machine designer, that can design a satisfactory data base structure for a set of information queries. Since this designer uses only a set of anticipated queries as input, the ouput is a functional design rather than an existential design.

3.1.2 Relation to programmer.

The automatic data base structure designer and the automatic information retrieval programmer of Chapter 2 form a package. Both programs use the HI-IQ query language. The automatic designer generates a data base structure that can be used as a data base description for the automatic programmer.

The programs are also similar because both translate from relational descriptions to access path descriptions.

3.1.3 Limitations.

This chapter is concerned with the design of data structures. Data structure design is only part of the overall problem of data base design. Some other design issues, not considered here, are decisions regarding data types, data base subdivision into areas, storage media, record ordering, record selection, and data protection. Also, data characteristics such as relative volumes, volatility of data, etc., that may affect data base structure design decisions have been ignored in this chapter. Ignoring these design issues and data characteristics will never result in data structures that are inconsistent with the set of queries, but it may lead to the design of inefficient data structures.

The automatic structure designer determines record content and record relationships (sets) and also suggests inversions and items to be used as calculated keys. The output of the automatic designer is not equivalent to a

complete DDL [CODASYL 1971a] description of a data base. Many statements necessary in a complete DDL description affecting record ordering, set ordering, selection mode, data privacy, data pictures, area content, etc., are not generated by the automatic data base structure designer.

3.2 Use of the Programmer.

The retrieval program generator described in the previous chapter can be used by the DBA during the data base design process. Since the program generator is data-free, program generation is solely a function of the data base description and the query. This "data-freeness" is an important difference between generalized interpreters and automatic programming. It is one of my reasons for using automatic programming rather than designing a generalized interpreter. (See section 2.3.) Data-freeness also permits use of the programmer during the design phase when a data base is not yet available.

Figure 3-1 illustrates the use of a programmer (human or machine) to aid in data structure design. The user communicates his needs (in the form of a query) to the programmer, the programmer obtains the proposed data base structure from the DBA and attempts to write a program. If the programmer is unsuccessful he communicates this to the DBA. Otherwise, he provides the DBA with the program. In either case the DBA can judge the validity of the proposed data base structure and make appropriate modifications. If any modifications are made, then the programmer tries again to create a program until a cycle is completed in which no data base structure modifications occur.

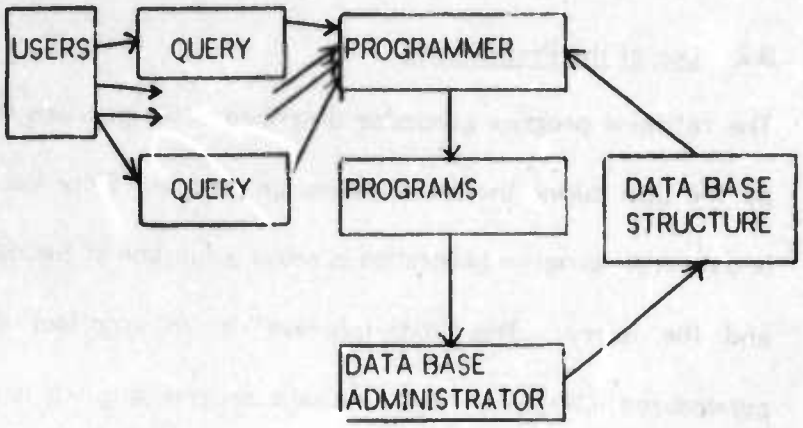


Figure 3-1. A simple user - programmer - DBA system.

This same process is then repeated for another query until all queries have been translated to satisfactory programs. Upon completion of this time consuming process the DBA can be reasonably confident that the proposed structure is sufficient for user needs. Note that the data base itself is not involved in this system. (Another feedback loop in the user-programmer-DBA system is closed after the data base has been created and the user obtains the results of the execution of the programs associated with his needs. At this point other adjustments become necessary, usually resulting in restructuring, discussed in Chapter 4.)

This technique is laborious and random, especially for the DBA. To make the process feasible, other lines of communication are necessary between the programmer and the DBA (to report causes of failure) and between the users

and the DBA (for naming conventions). An algorithm for finding the initial data base structure must also be specified.

Human programmers cannot be used for this method of design because they would take too long and the cost of writing the programs for testing each design would be too high. This restriction does not apply to the machine programmer discussed in Chapter 2. However, I have not designed a data base structure using the automatic programmer in the process described above, but consideration of this process led me to automate the data base structure design process in its entirety. This is the subject of the next section.

3.3 Automatic Data Structure Design.

3.3.1 Overview.

The implemented designer generates a data structure design in one pass as illustrated in Figure 3-2. This system is not iterative as was the process discussed for Figure 3-1.

Structure design requires two phases. In the first phase all queries are scanned, and for each query a set of assertions is generated that describe the structural concepts of the data base necessary for the query. A particular assertion is a constraint on the design, but there may be many designs which satisfy that assertion. The second phase tries to find a design that satisfies all of the assertions.

The two phases are illustrated in Figure 3-2 as follows:

(a) Generate assertions
(first three steps in Figure 3-2)

(b) Record and network construction
(last four steps in Figure 3-2)

The program for phase 1 is almost identical to the Request Handler described in Chapter Two. Since the HI-IQ language is used by both the designer and the program generator, the query interpreter will be the same for both. Furthermore, each assertion made in the first design phase corresponds to a structural concept that the program generator would require in the defined data

structure.

The designer permits interruption while generating assertions. This allows a structure to be designed from a subset of the queries with a later continuation of the design process from the point of interruption. The DBA can therefore use the system to quickly determine the impact on the data base structure of a particular subset of queries.

The actual design process occurs in the fourth and fifth boxes of the chart in Figure 3-2. In these steps the system tries to derive a network structure design that is consistent with the assertions.

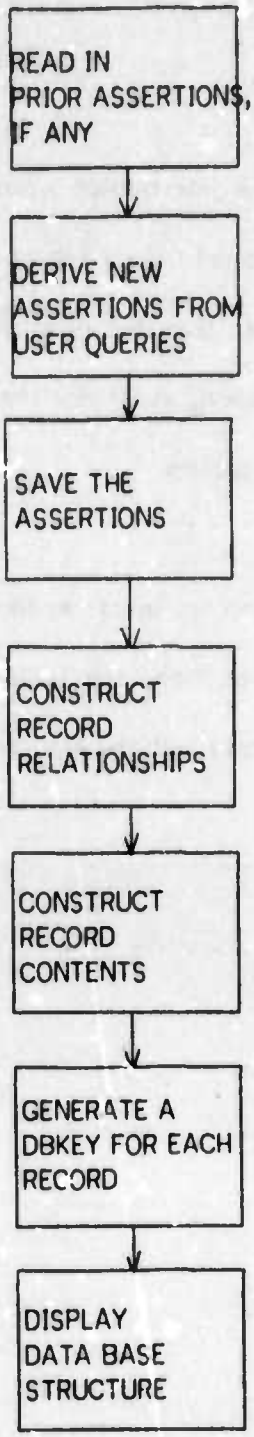


Figure 3-2. Data base structure designer; general program flow.

3.4 Defining Item Names.

Since several users may be generating the queries, there are two problems related to item (attribute) names. Different users might use the same name for different attributes or they might use different names when referring to the same attribute.

In addition, misspelling can occur. A design system has difficulty with these problems because it has no external corroboration for verification of names.

There are several ways of handling this problem. I have chosen the last of these approaches:

a) Indicate the caveat: Errors in data base structure may result from errors in naming items and records or from the use of identical names for different items and records.

b) Interactive corroboration: Provide for interaction between the machine designer and the DBA whenever a name is encountered for the first time or in any other situations where errors might go undetected without additional corroboration.

c) Definitive corroboration: Require a definition list of names prior to beginning the design process.

3.5 Generating Assertions.

The automatic designer first generates assertions regarding the data base structure that is to be designed. These assertions are used to direct and constrain the eventual design process.

The assertions made in this phase concern item and record relationships. Although the query may contain commands such as COUNT, ANY, REPEAT, etc., these commands are ignored except to note that the record mentioned in the context of such a command must be hierarchically below the record mentioned prior to that command. Similarly, the designer makes no distinction between the "CONDITIONS FOR RETRIEVAL" and the "ITEMS OR STATS" inputs for each record. Both inputs are simply scanned for names which are not commands. These names are then checked against the master name list and if found, assumed to be items.

That the designer ignores these commands should give some indication of its limitations. A better system would consider some of the aforementioned usages to (for example) design an ordered set.

Assertions are made wherever the programmer would have checked for the presence of certain relationships in the defined structure. There are only three such assertions. These are listed along with their meanings in Figure 3-3. Succeeding paragraphs will discuss the generation of these assertions; it should

be noted that duplicate assertions are never generated.

ABOVE (A,B)
Record A is hierarchically above record B.

INORABOVE (I,A,B)
Item I must be contained in record A or in a record hierarchically above record A unless records A and B form a confluent hierarchy, in which case item I must be contained in or above the record which forms the base of the confluent hierarchy.

CALCPORT (I)
Item I has been used in an equivalency test and may therefore be suitable for use as a calculated key.

Figure 3-3. Assertions and their meanings.

3.5.1 The ABOVE assertion.

An ABOVE assertion is generated for every record name encountered in a query except for the top level record of the query. Thus if A is the top level record and B and C are second level records in a particular query, then two assertions, ABOVE(A,B) and ABOVE(A,C) are made. If D is a third level record and is mentioned within the second level context of record B then the third assertion made is ABOVE(B,D). The actual design process will treat these assertions as constraints so that ABOVE(A,B) can be satisfied in one of three ways:

a) Records A and B are top level members of the same confluent hierarchy

b) Existing hierarchies provide an upward path from record B to record A (transitivity)

c) Records A and B are contained in a set with A as owner and B as member.

3.5.2 The INORABOVE assertion.

Items can be mentioned in a query both as intended for display or for use in a conditional test. In either case they are always mentioned within the context of a particular record. The principle of downward attribute migration lets the system conclude that the items must be physically located within or hierarchically above the context record.

For those unfamiliar with the concept of confluent hierarchies, an example of such a structure is given on the left in Figure 3-4. The TREATMENT record is the base record of this structure because it is hierarchically below all other records. This structure is also discussed in section 1.3, Figure 1-3. Since any of the records included in a confluent hierarchy can occur hierarchically above any of the other records, such a structure can be processed "both ways". The query context determines which way the confluent hierarchy is being processed. It is for this reason that the INORABOVE assertion includes a description of the query context, described with records A and B.

Figure 3-4 illustrates the effect of query context. It is perfectly legal for a query to refer to the unique DIAGNOSIS associated with a particular PATIENT record if the query context is the DOCTOR-PATIENT hierarchy, even though the actual data base structure has the DIAGNOSIS item below the PATIENT record.

This is legal since the relationship can be reversed because of the confluency of the DOCTOR and PATIENT records. Because of this confluent hierarchy it is possible to associate many DIAGNOSIS records with a PATIENT in one context and to associate a unique DIAGNOSIS in another context.

Since it is not known at the time the assertions are made if a particular context hierarchy will be part of a confluent hierarchy, the system must indicate the hierarchical context, not just the context record, when describing the use of a particular item. Thus, for the previous example, the assertion would be: INORABOVE(DIAGNOSIS,PATIENT,DOCTOR). To the second phase, this assertion will mean "DIAGNOSIS must be contained within or above the PATIENT record unless the PATIENT and DOCTOR records exist in a confluent hierarchy in which case the DIAGNOSIS item must be contained within or above the lowest or base record of the confluent hierarchy."

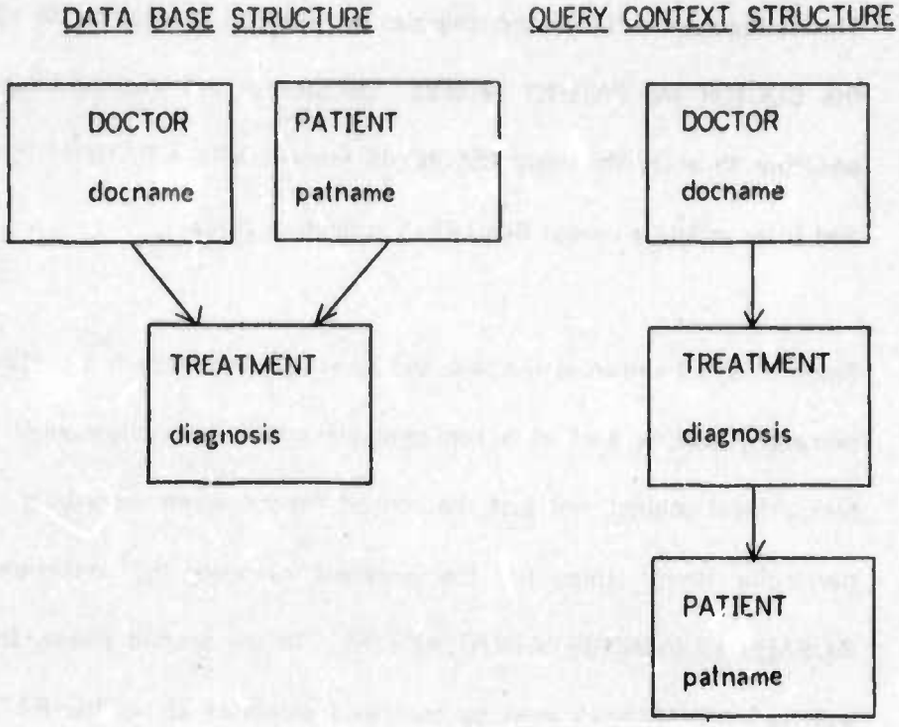


Figure 3-4. Transformation of a confluent hierarchy within the query context.

3.5.3 The CALCPORT assertion.

The designer does not use the CALCPORT assertion. It is generated to aid the DBA in determining which items should be used as calculated keys or in file inversions. CALCPORT(I) states that item I has been used in a test of the form (I EQ...), and that possible use of I as a calculated key is suggested.

3.5.4 Remark on attribute migration.

In the first two chapters the concept of attribute migration was introduced.

3.5 Generating Assertions.

Since the data base structure designer ignores all statistical commands it need be concerned only with downward attribute migration. If attribute (item) I occurs in the context of record A then, because of downward attribute migration, the system can conclude that item I can only be physically stored with record A or any record which is hierarchically above record A. Thus from every direct item-record association the system derives a lower bound on the physical location of that item within the data base structure. If the system always physically locates an item at the highest lower bound encountered so far, then an item's physical location will tend to migrate upward in the structure as the structure becomes more constrained during the generation of assertions. Downward attribute migration results in upward physical location migration during the design process.

3.6 Designing Record Relationships.

The process of designing record relationships is illustrated in Figure 3-5. Two types of logical relationships are recognized: direct hierarchies and confluencies. Since a confluency can be expressed with two or more direct hierarchies, the final design will specify only direct hierarchies. Each such hierarchy is defined with a HIERARCHYGROUP assertion. Each box in the flow is actually a loop which attempts to erase ABOVE assertions and continues to do so until an iteration occurs in which no assertion is erased.

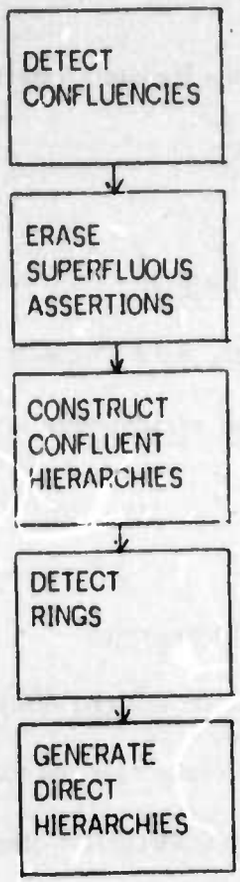


Figure 3-5. Construction of record relationships.

The principal input to this phase of the design process is the ABOVE assertions generated by the Request Handler. When the record relationship design process has been completed, there are no remaining ABOVE assertions. A brief discussion of each step in Figure 3-5 follows.

3.6.1 Detect confluencies.

If ABOVE(A,B) and ABOVE(B,A) have been asserted then records A and B exist in

a confluency. The system therefore asserts CONFLUENCY(A,B) and erases both of the ABOVE assertions.

3.6.2 Eliminate redundant ABOVE.

If ABOVE(A,B), ABOVE(B,C), and ABOVE(A,C) have been asserted, then ABOVE(A,C) is redundant (and can be erased) since it is derivable from the remaining two assertions.

3.6.3 Construct confluent hierarchies.

If CONFLUENCY(A,B), ABOVE(A,C) and ABOVE(B,C) have all been asserted, then the confluency has been captured in the confluent hierarchy defined by the two ABOVE assertions and the CONFLUENCY assertion is simply erased.

If CONFLUENCY(A,B) has been asserted, but no record C exists as above, then the system constructs a new (artificial) record, D, and asserts ABOVE(A,D) and ABOVE(B,D). The CONFLUENCY assertion is erased as above.

In both cases, the system will alter all INORABOVE(I,A,B) assertions to INORABOVE(I,C,-) or INORABOVE(I,D,-) respectively. This step recognizes that all items referenced within the context of the (A,B) confluency must be contained in or above the base record (C or D respectively) of the confluent hierarchy.

The reason for processing confluencies in two steps, separated by the step to erase redundant assertions, is best illustrated by examples.

Consider the following set of assertions:

ABOVE(A,B) ABOVE(B,A)

ABOVE(A,C) ABOVE(B,C)

Either of the latter two assertions is redundant (not both). Assume ABOVE(B,C) is erased before the detection of confluencies. Then the following assertions remain:

ABOVE(A,B) ABOVE(B,A)

ABOVE(A,C)

Now the system can not discover that C is the base record of the confluent hierarchy.

Although postponing the erasure of redundant assertions to follow the generation of confluent hierarchies will not affect the design of record relationships, it will have an adverse effect on the later design of record contents.

Again, consider an example. If

ABOVE(A,B) ABOVE(B,A)

ABOVE(A,C) ABOVE(B,C)

ABOVE(A,D) ABOVE(B,D)

ABOVE(C,D)

INORABOVE(I,A,B)

have all been asserted, then postponing the erasure of ABOVE(A,D) and ABOVE(B,D) may result in the detection of record D as the base record of the confluency. This in turn will result in the incorrect assertion INORABOVE(I,D,-).

3.6.4 Detect rings.

If ABOVE(A,B), ABOVE(B,C) and ABOVE(C,A) have all been asserted, then the system asserts RING(A,B). This assertion has no further impact on the design process, but it tells the DBA that the data structure contains a circular sub-structure. Such a sub-structure, although legal, is highly unlikely, and its occurence will usually signal the presence of an error.

3.6.5 Generate direct hierarchies.

This is the last step of Figure 3-5, and it transforms all remaining ABOVE assertions to HIERARCHYGROUP assertions. Each HIERARCHYGROUP assertion defines a DBTG "set" and therefore requires a unique name. The system generates such names in the form SYMnn (00<nn<100).

3.7 A Frame for Record Relationship Design.

The structure designer is defined to the APG with a Frame of rules (Figure 3-6). The Frame is compiled to Micro-Planner theorems by the APG system. (See section 2.2.) These Micro-Planner theorems form the basis of the data base structure designer's decision subsystem. Although the APG compiler is used, the resulting program is not a procedure generator, but rather a declarative generator. Use of the APG compiler to compile programs which are not procedure generators requires the implementation of a new rule of inference.

This new rule, called an assertion rule (denoted by S5), permits assertions to be made from pre-conditions without procedure generation. An S5 rule is not equivalent to a logical implication since the post-condition is asserted to be true and may contradict the pre-condition. When such a contradiction exists, the rule erases those assertions that are true in the pre-condition and false in the post-condition. These erasures help to keep the set of assertions as concise as possible.

Type S3 rules are also used to define the design process. This type of rule is described in detail in Chapter Two. Type S3 rules are the equivalent of implication in logic.

Either type of rule is defined with a pre- and post-condition such that the truth of the pre-condition implies the truth of the post-condition. In Figure 3-6, the

post-condition is differentiated from the pre-condition with a right arrow (→).

RULE NAME	TYPE	PRE AND POST CONDITIONS
DETECT CONFLUENCIES	S5	ABOVE(A,B)∧ABOVE(B,A) → ¬ABOVE(A,B)∧¬ABOVE(B,A)∧ CONFLUENCY(A,B)
ELIMINATE REDUNDANT ABOVE	S5	ABOVE(A,B)∧INDABOVE(A,B) → ¬ABOVE(A,B)
CONSTRUCT CONFLUENT HIERARCHIES	S5	CONFLUENCY(A,B)∧ [COMBOT(A,B,C)∨=(C,(GENSYM))]∧ SWITCHINORABOVE(A,B,C)∧ SWITCHINORABOVE(B,A,C) → ¬CONFLUENCY(A,B)∧ ABOVE(A,C)∧ABOVE(B,C)
DETECT RINGS	S5	ABVUPLINK(A,B)∧ABVUPLINK(B,A) → RING(A,B)
GENERATE DIRECT HIERARCHIES	S5	ABOVE(A,B) → ¬ABOVE(A,B)∧ HIERARCHYGROUP(A,B,(GENSYM))
COMMON BOTTOM	S?	[ABVUPLINK(C,A)∧ABVUPLINK(C,B)]∨ [CONFLUENCY(C,D)∧ABVUPLINK(D,B)∧ [=(C,A)∨ABVUPLINK(C,A)]] → COMBOT(A,B,C)

Figure 3-6 is continued on the next page.

RULE NAME	TYPE	PRE AND POST CONDITIONS
LINK UP USING ABOVE	S3	ABOVE(B,A)∨INDABOVE(B,A) → ABVUPLINK(A,B)
INDIRECTLY ABOVE	RECURSIVE S3	ABOVE(A,C)∧¬=(B,C)∧ [ABOVE(C,B)∨INDABOVE(C,B)] → INDABOVE(A,B)
CONFLUENCY EQUIVALENCE	S3	CONFLUENCY(B,A) → CONFLUENCY(A,B)
EXCHANGE ALL ITEMS IN CONFLUENT HIERARCHY	RECURSIVE S3	¬INORABOVE(I,A,B)∨ [INORABOVE(I,A,B)∧EXCHANGE(I,A,B,C)∧ SWITCHINORABOVE(A,B,C)] → SWITCHINORABOVE(A,B,C)
EXCHANGE AN ITEM IN CONFLUENT HIERARCHY	S5	INORABOVE(I,A,B) → EXCHANGE(I,A,B,C)∧ ¬INORABOVE(I,A,B)∧ INORABOVE(I,C,A)

Figure 3-6. Rules for generating record relationships.

3.7.1 Assertions.

The pre- and post-conditions are defined in terms of assertions and the usual logical notation, ∧, ∨, and ¬. Determination of the truth of the post-condition is accomplished by evaluating some, if not all, of the assertions in the pre-condition. A particular assertion is evaluated to true (false) if it is true

(false) in the present state, or by the evaluation of the pre-condition of a rule which has the particular assertion as a post-condition.

The not sign (¬) preceeding an assertion in a post-condition causes that assertion to be removed from the current state.

(GENSYM) causes a new symbol to be generated in the form of SYMnn. Thus, "=(C,(GENSYM))" causes the variable C to be bound to a new symbol.

A semantic interpretation of the assertions is given in Figures 3-3 and 3-7.

ASSERTION	MEANING
CONFLUENCY(A,B)	Records A and B form a confluency.
COMBOT(A,B,C)	Records A and B are both directly or indirectly above record C.
SWITCHINORABOVE(A,B,C)	All INORABOVE(I,A,B) have been changed to INORABOVE(I,C,A).
INDABOVE(A,B)	Record A is indirectly but not directly above record B (uses transitivity of ABOVE).
HIERARCHYGROUP(A,B,S)	Record A is the owner and record B is the member of set S.
ABVUPLINK(A,B)	Record B is directly or indirectly above record A.
EXCHANGE(I,A,B,C)	INORABOVE(I,A,B) has been converted to INORABOVE(I,C,A).
DBKEY(A,I)	I is the data base key for record A.
CONTAINS(A,I)	Record A physically contains item I.
UPLINK(A,B)	Record B is physically above record A. (Uses HIERARCHYGROUP rather than ABOVE.)
COMTOP(A,B,C)	Record C is physically above both records A and B.
RING(A,B)	Structure contains at least one Ring and it contains records A and B.
=(A,B)	A has been set equal to B.

Figure 3-7. Assertions and their interpretations.

3.7.2 Using the rules.

The first five rules in Figure 3-6 correspond to the five steps in Figure 3-5 described earlier. The remaining rules are used to determine the values of some of the assertions used in the pre-conditions of the five principal rules.

The five principal rules are each invoked repetitively by a control processor to

accomplish the processing defined in Figure 3-5. For example, DETECT CONFLUENCIES is invoked with the goal ¬ABOVE(A,B). If this goal is successful (e.g., a confluency is detected), the rule is re-invoked with the same goal until the goal is not achieved.

The APG system was extended to include the type S5 axiom in order to accomodate this type of processing. Manipulation of the assertions, including erasure thereof, is necessary to insure termination of processing. Without such modification, a rule which is true once remains true forever.

3.7.3 An example.

Figure 3-8 illustrates the change in the assertions after applications of the first three rules.

BEFORE	AFTER
ABOVE(DOCTOR,PATIENT)	ABOVE(PATIENT,TREATMENT)
ABOVE(PATIENT,DOCTOR)	ABOVE(DOCTOR,TREATMENT)
ABOVE(DOCTOR,TREATMENT)	ABOVE(DOCTOR,SYM01)
ABOVE(PATIENT,TREATMENT)	ABOVE(HOSPITAL,SYM01)
ABOVE(DOCTOR,HOSPITAL)	
	INORABOVE(DIAGNOSIS,TREATMENT,DOCTOR)
ABOVE(HOSPITAL,DOCTOR)	INORABOVE(OFFICENO,SYM01,HOSPITAL)
INORABOVE(DIAGNOSIS,DOCTOR,PATIENT)	
INORABOVE(OFFICENO,HOSPITAL,DOCTOR)	

Figure 3-8. Assertions before and after application of the first three rules.

This example contains two confluencies:

DOCTOR ↔ PATIENT
HOSPITAL ↔ DOCTOR

which are detected by the first rule. This results in the erasure of four ABOVE assertions and the assertion of two CONFLUENCY's.

There are no redundant ABOVE assertions, so none are erased using the second rule.

The third rule constructs confluent hierarchies. It uses the COMMON BOTTOM rule to prove:

COMBOT(DOCTOR,PATIENT,TREATMENT)

and the last two rules of Figure 3-6 to change

INORABOVE(DIAGNOSIS,DOCTOR,PATIENT) to

INORABOVE(DIAGNOSIS,TREATMENT,-). The second confluency is similarly processed, except that a common bottom for HOSPITAL and DOCTOR is not found so one is created (SYM01).

3.7.4 Remark on usual data structures.

Figure 3-9 shows three interesting conflicting structures and their conversion to confluent hierarchies. The X's stand for records generated by the system. Single arrows indicate ABOVE relationships; A→B is equivalent to ABOVE(A,B). Double arrows indicate confluencies; A↔B is equivalent to CONFLUENCY(A,B).

Note that due to the Confluency Equivalence rule this is also equivalent to CONFLUENCY(B,A).

The first case is interesting because any record is below the other two yet if this record is used as the base for a confluent hierarchy then only two confluent hierarchies will result in the final structure, as is illustrated in the incorrect sequence. The Common Bottom rule (with one exception) excludes confluencies to find a COMBOT.

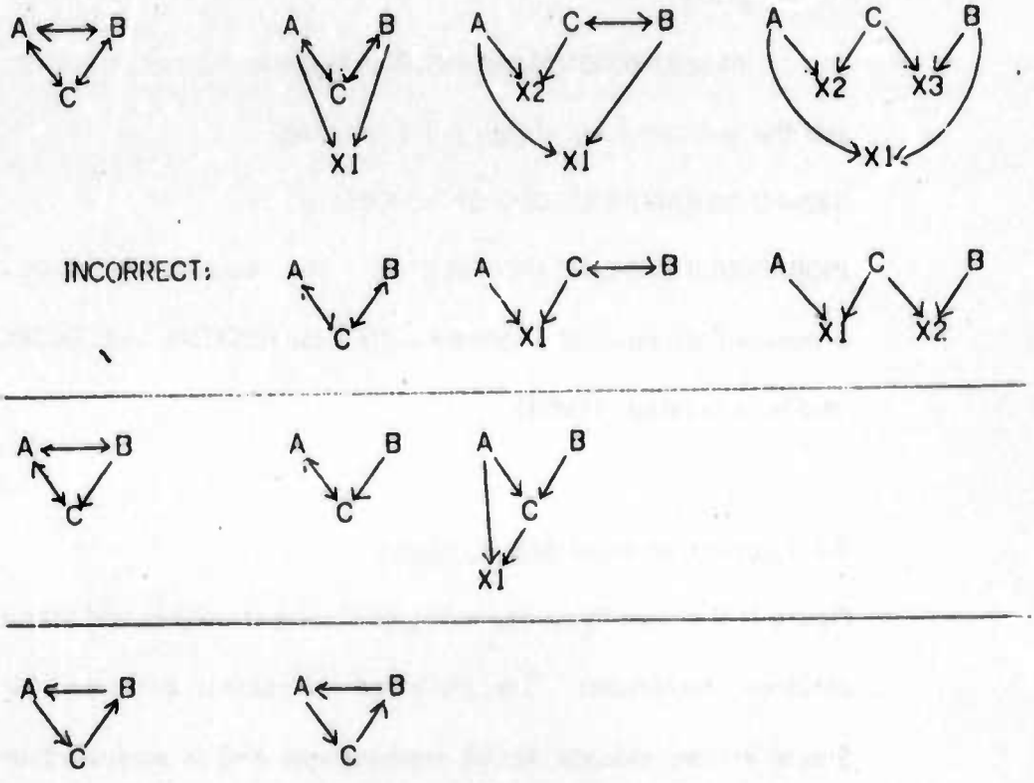


Figure 3-9. Some interesting structure transformations.

The exception to the Common Bottom rule is illustrated in the second case in Figure 3-9. The second case differs from the first only in that one of the confluencies has been removed. Since the A↔C confluency will eventually result in the detection or creation of a record, say D, which is below record C, D will also be below record B. In general, a "common bottom" can be either a record or another confluency, but the links used to establish it must all be ABOVE assertions. Note that if the A↔C confluency were processed first, the same final structure would result, but the X1 record would be generated in the first rather than the last step.

The third case is interesting because, although valid, it is probably the result of error or omission in the set of queries. Most likely one of the hierarchies either was erroneously referenced or is actually a confluency but was never referenced in the other direction. The system does not alter this structure, but the final data base structure design will contain assertion of the form RING(A,B), warning the DBA that at least one ring structure exists and that it contains, among others, records A and B. The rule for detecting a RING structure is given in Figure 3-6.

3.8 Designing Record Contents.

Processing the INORABOVE assertions to provide CONTAINS assertions for the data base structure is similar to the processing of the ABOVE assertions to produce the HIERARCHYGROUP assertions. Figure 3-10 gives the general flow, and Figure 3-11 contains the rules. The first three rules correspond to the steps in Figure 3-10. Each box in the flow is actually a loop which attempts to erase INORABOVE assertions and continues to do so until an iteration occurs in which no assertion is erased.

The process illustrated in Figure 3-10 ignores the third argument in the INORABOVE assertions, since all processing associated with confluencies was done when confluencies were transformed to confluent hierarchies.

Figure 3-10. Determination of record content - general flow.

RULE NAME	TYPE	PRE AND POST CONDITIONS
ELIMINATE REDUNDANT ASSERTIONS	S5	INORABOVE(I,A,C)∧INORABOVE(I,B,D)∧ [[¬=(A,B)∧UPLINK(A,B)]∨ [=(A,B)∧¬=(C,D)]] → ¬INORABOVE(I,A,C)
FIND COMMON OWNERS	S5	INORABOVE(I,A,C)∧INORABOVE(I,B,D)∧ ¬=(A,B)∧COMTOP(A,B,E) → ¬INORABOVE(I,A,C)∧INORABOVE(I,B,D)∧ INORABOVE(I,E,A)
CREATE CONTAINS ASSERTIONS	S5	INORABOVE(I,A,B) → ¬INORABOVE(I,A,B)∧CONTAINS(A,I)
LINK UP	RECURSIVE S3	HIERARCHYGROUP(B,A,S)∨ [HIERARCHYGROUP(C,A,S)∧UPLINK(C,B)] → UPLINK(A,B)
COMMON TOP 1	S3	UPLINK(A,C)∧UPLINK(B,C) → COMTOP(A,B,C)
COMMON TOP 2 (IF TOP 1 FAILS)	S5	=(C,(GENSYM)) → COMTOP(A,B,C)∧ HIERARCHYGROUP(C,A,(GENSYM))∧ HIERARCHYGROUP(C,B,(GENSYM))

Figure 3-11. Rules for establishing record contents.

3.8.1 Eliminate redundant assertions.

The system first eliminates redundant INORABOVE assertions. INORABOVE(I,A,-) is redundant if item I is also specified to be in or above another record B, and B is physically above A (using transitivity). This makes INORABOVE(I,A,C) redundant because INORABOVE(I,B,-) implies that I is above A.

3.8.2 Find common owners.

If, following the elimination of redundancies, more than one INORABOVE assertion remains for item I, then some record, say X, must be found which connects the separate substructures in which I is referenced. COMTOP finds such a record, or generates one if it does not exist. In either case all INORABOVE assertions referencing I will be replaced by INORABOVE(I,X,-).

When record generation occurs, there will be a COMTOP assertion in the final data base structure to indicate to the DBA the reason for the existence of the generated record. A COMTOP assertion, as does a RING assertion, probably signals an erroneous use or omission in the set of queries used by the structure designer.

3.8.3 Create CONTAINS assertions.

After all appropriate Common Top processing has been done there will be exactly one INORABOVE (I,A,-) for each item I, indicating that the item should be contained in Record A. All remaining assertions are therefore directly

transformed by the third rule to CONTAINS(A,I).

3.9 Example.

Figures 3-12 through 3-19 give a detailed example of the design process. The user's queries are given in Figures 3-12 to 3-17. This is not a complete set of queries for an anticipated implementation so the resulting data base structure will probably not be complete. In each query, the system prompts are underlined to distinguish them from user replies. For expository purposes, these queries are somewhat contrived.

Figure 3-18 contains the assertions generated in the first phase. Figure 3-18 also indicates which additional assertions resulted from each query when the queries were processed in numerical order. Had the queries been processed in a different order, the resulting set of assertions would have been the same but the subset of assertions added by a particular query might have been different.

Figure 3-19 contains the data base structure design as output by the system. The COMTOP assertion for system-created record SYM12 (containing DOCNAME) will lead the DBA to discover an error in the first query where DOCNAME is indicated within the context of the PATIENT record and not in the context of a possible confluent hierarchy. (PATIENT is the top level record of the first query.)

A representation of the resulting data base structure using data structure diagrams is given in Figure 3-20. The six queries used were initially created to

debug the automatic programmer of Chapter 2 using the data base of Figure 2-28. It is therefore interesting to note the close similarity between the human designed data base of Figure 2-28 and the machine design of Figure 3-20. Primary differences are the lack of inversion for the PATIENT record on RACE and SEX (although the system suggests SEX as a CALCPORT), the non-existence of a confluent hierarchy for DOCTOR and HOSPITAL records, and the omission of a HOSPITAL - PATIENT hierarchy.

Since the queries consider a HOSPITAL-DOCTOR but not a DOCTOR-HOSPITAL hierarchy, the confluency is not present in the designed data base structure. Similarly, none of the queries reference a HOSPITAL-PATIENT hierarchy, so this structure is also not included.

3.9 Example.

```
PRIMARY RECORD (MAIN)
*PATIENT
CONDITIONS FOR RETRIEVAL
*(SEX EQ 'M')
*NIL
ITEMS OR STATS TO BE DISPLAYED
*DOCNAME
*REPEAT
  PRIMARY RECORD (REPEAT)
  *TREATMENT
  CONDITIONS FOR RETRIEVAL
  *(COUNT GE 3)
    PRIMARY RECORD (COUNT)
    *ORDERS
    CONDITIONS FOR RETRIEVAL
    *NIL
  *NIL
  ITEMS OR STATS TO BE DISPLAYED
  *DIAGNOSIS
  *NIL
*NIL
```

Figure 3-12. Query number one: "For all male patients display the associated doctor's name and all diagnoses for which at least three orders have been prescribed."

3.9 Example.

```
PRIMARY RECORD (MAIN)
*DOCTOR
CONDITIONS FOR RETRIEVAL
*(COUNT GT 5)
   PRIMARY RECORD (COUNT)
   *PATIENT
   CONDITIONS FOR RETRIEVAL
   *(SEX EQ 'M')
   *NIL
*NIL
ITEMS OR STATS TO BE DISPLAYED
*DOCNAME
*REPEAT
   PRIMARY RECORD (REPEAT)
   *TREATMENT
   CONDITIONS FOR RETRIEVAL
   *(SEX EQ 'M')
   *NIL
   ITEMS OR STATS TO BE DISPLAYED
   *DIAGNOSIS
   *PATNO
   *PATNAME
   *NIL
*NIL
```

Figure 3-13. Query number two: "For all doctors having more than five patients, display the doctor's name and the diagnosis, patient's name and number for every male patient being treated by that doctor."

3.9 Example.

```
PRIMARY RECORD (MAIN)
*PATIENT
CONDITIONS FOR RETRIEVAL
*(COUNT EQ 6)
   PRIMARY RECORD (COUNT)
   *DOCTOR
   CONDITIONS FOR RETRIEVAL
   *NIL
*(PATNO LT 6000)
*NIL
ITEMS OR STATS TO BE DISPLAYED
*PATNAME
*PATNO
*REPEAT
   PRIMARY RECORD (REPEAT)
   *BILLENTRY
   CONDITIONS FOR RETRIEVAL
   *(AMOUNT GT 100)
   *NIL
   ITEMS OR STATS TO BE DISPLAYED
   *AMOUNT
   *CODE
   *NIL
*HOSPNAME
*NIL
```

Figure 3-14. Query number three: "For all patients with exactly 6 doctors and who have a patient number less than 6000, display the patient's name, number, hospital and the amount and code for all bills over $100."

```
PRIMARY RECORD (MAIN)
*HOSPITAL
CONDITIONS FOR RETRIEVAL
*ANY
  PRIMARY RECORD (ANY)
  *DOCTOR
  CONDITIONS FOR RETRIEVAL
  *(DOCNAME EQ 'SMITH')
  *ANY
    PRIMARY RECORD (ANY)
    *TREATMENT
    CONDITIONS FOR RETRIEVAL
    *(DIAGNOSIS EQ 'TUBERCULOSIS')
    *NIL
  *NIL
*NIL
ITEMS OR STATS TO BE DISPLAYED
*HOSPNO
*NIL
```

Figure 3-15. Query number four: "Display the hospital number for all hospitals that have a doctor named Smith who has diagnosed tuberculosis."

3.9 Example.

```
PRIMARY RECORD (MAIN)
*DOCTOR
CONDITIONS FOR RETRIEVAL
*ANY
   PRIMARY RECORD (ANY)
   *TREATMENT
   CONDITIONS FOR RETRIEVAL
   *(DIAGNOSIS EQ 'PNEUMONIA')
   *NIL
*(AVE GT 3)
   PRIMARY RECORD (AVE)
   *TREATMENT
   CONDITIONS FOR RETRIEVAL
   *NIL
   ITEMS OR STATS FOR AVE
   *COUNT
      PRIMARY RECORD (COUNT)
      *ORDERS
      CONDITIONS FOR RETRIEVAL
      *NIL
   *NIL
*NIL
ITEMS OR STATS TO BE DISPLAYED
*DOCNAME
*COUNT
   PRIMARY RECORD (COUNT)
   *ORDERS
   CONDITIONS FOR RETRIEVAL
   *NIL
*NIL
```

Figure 3-16. Query number five: "For all doctors who have diagnosed pneumonia or whose average number of orders per treatment is less than three, display the doctor's name and a count of the number of orders prescribed."

3.9 Example.

```
PRIMARY RECORD (MAIN)
*PATIENT
CONDITIONS FOR RETRIEVAL
*(PATNO EQ 10)
*NIL
ITEMS OR STATS TO BE DISPLAYED
*MIN
   PRIMARY RECORD (MIN)
   *ORDERS
   CONDITIONS FOR RETRIEVAL
   *NIL
   ITEMS OR STATS FOR AVE
   *ORDNO
   *NIL
*NIL
```

Figure 3-17. Query number six: "For the patient with the number 10, display the smallest order number prescribed for that patient."

3.9 Example.

REQ	ASSERTION
6	CALCPORT (PATNO)
4	CALCPORT (DIAGNOSIS)
4	CALCPORT (DOCNAME)
1	CALCPORT (SEX)
6	ABOVE (PATIENT ORDERS)
5	ABOVE (DOCTOR ORDERS)
4	ABOVE (HOSPITAL DOCTOR)
3	ABOVE (PATIENT BILLENTRY)
3	ABOVE (PATIENT DOCTOR)
2	ABOVE (DOCTOR TREATMENT)
2	ABOVE (DOCTOR PATIENT)
1	ABOVE (TREATMENT ORDERS)
1	ABOVE (PATIENT TREATMENT)
6	INORABOVE (ORDNO ORDERS PATIENT)
4	INORABOVE (HOSPNO HOSPITAL PORT)
4	INORABOVE (DOCNAME DOCTOR HOSPITAL)
3	INORABOVE (HOSPNAME PATIENT PORT)
3	INORABOVE (CODE BILLENTRY PATIENT)
3	INORABOVE (AMOUNT BILLENTRY PATIENT)
3	INORABOVE (PATNAME PATIENT PORT)
3	INORABOVE (PATNO PATIENT PORT)
2	INORABOVE (PATNAME TREATMENT DOCTOR)
2	INORABOVE (PATNO TREATMENT DOCTOR)
2	INORABOVE (DIAGNOSIS TREATMENT DOCTOR)
2	INORABOVE (SEX TREATMENT DOCTOR)
2	INORABOVE (DOCNAME DOCTOR PORT)
2	INORABOVE (SEX PATIENT DOCTOR)
1	INORABOVE (DIAGNOSIS TREATMENT PATIENT)
1	INORABOVE (DOCNAME PATIENT PORT)
1	INORABOVE (SEX PATIENT PORT)

Figure 3-18. Assertions generated from queries 1 - 6.

3.9 Example.

```
COMTOP (PATIENT DOCTOR SYM12)
CALCPORT (SEX)
CALCPORT (DOCNAME)
CALCPORT (DIAGNOSIS)
CALCPORT (PATNO)
HIERARCHYGROUP (SYM12 PATIENT SYM15)
HIERARCHYGROUP (SYM12 DOCTOR SYM13)
HIERARCHYGROUP (HOSPITAL DOCTOR SYM10)
HIERARCHYGROUP (PATIENT BILLENTRY SYM08)
HIERARCHYGROUP (DOCTOR TREATMENT SYM06)
HIERARCHYGROUP (TREATMENT ORDERS SYM04)
HIERARCHYGROUP (PATIENT TREATMENT SYM02)
EXCHANGE (PATNO PATIENT DOCTOR TREATMENT)
EXCHANGE (SEX PATIENT DOCTOR TREATMENT)
DBKEY (ORDERS SYM23)
DBKEY (BILLENTRY SYM22)
DBKEY (TREATMENT SYM21)
DBKEY (DOCTOR SYM20)
DBKEY (PATIENT SYM19)
DBKEY (HOSPITAL SYM18)
DBKEY (SYM12 SYM17)
CONTAINS (ORDERS ORDNO)
CONTAINS (HOSPITAL HOSPNO)
CONTAINS (PATIENT HOSPNAME)
CONTAINS (BILLENTRY CODE)
CONTAINS (BILLENTRY AMOUNT)
CONTAINS (PATIENT PATNAME)
CONTAINS (PATIENT PATNO)
CONTAINS (TREATMENT DIAGNOSIS)
CONTAINS (PATIENT SEX)
CONTAINS (SYM12 DOCNAME)
```

Figure 3-19. Generated data base structure.

3.9 Example.

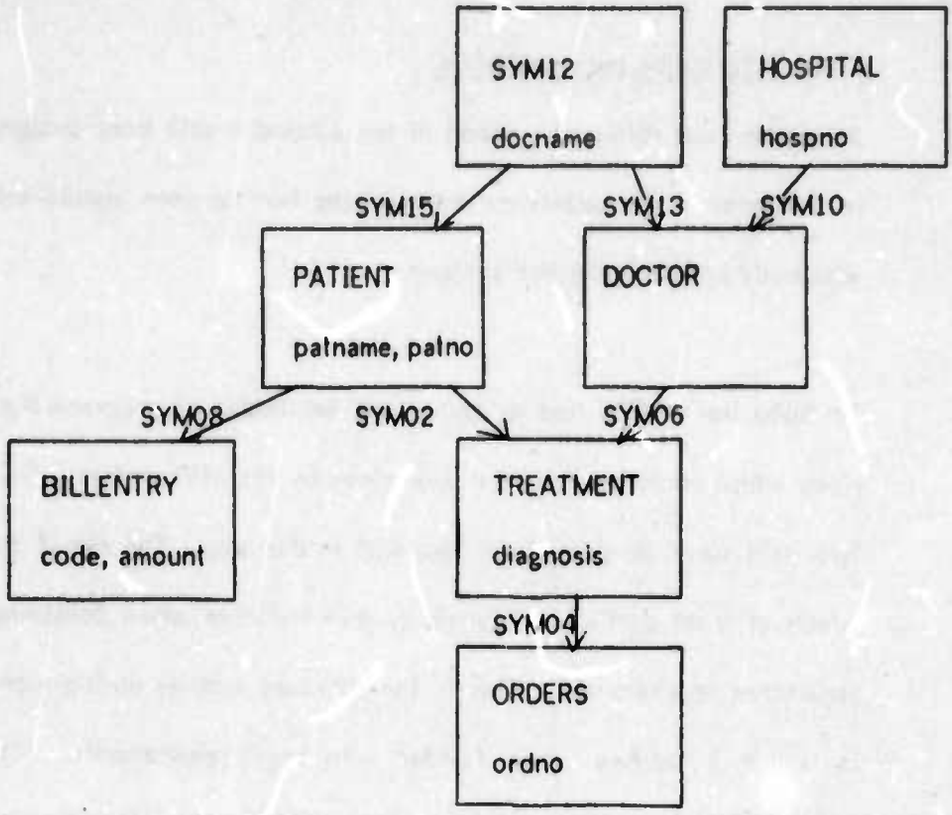


Figure 3-20. Data Structure diagram for Figure 3-19.

3.10 Alternative Implementation.

An earlier and different version of the automatic data base designer was also implemented. The difference between the two versions should interest those who build knowledge based systems.

To build the DMLP I had to capture my knowledge of programming in a set of rules which could be used and understood by the APG system. The first of the two data base designers was also built in this way. The result was a system which at least partially modelled my own behavior when designing data base structures: evolutionary. That is, the structure evolves during a period of time in which I become more familiar with user requirements. This has the advantage that at all times my current understanding of user requirements is captured in the current data structure.

Implementing this strategy in an automatic designer leads to some difficulties. These difficulties arise because as the structure evolves it is frequently necessary to make assumptions or to choose from several equally viable alternatives. This makes the process rather ad hoc, especially later in the design when further constraints require that some earlier assumptions and choices be undone. Programming this strategy therefore requires a precognizance of all possible occurences of the situations where ad hoc decisions are necessary.

It became quickly obvious that such pre-planning is very difficult. There were frequent additions and modifications to the system for handling conditions that had not been forseen. Furthermore it was impossible to prove any consistency in system outputs.

The second implementation, which grew out of the frustrations from the first differs in that the design does not evolve. The second system puts information extracted from queries into an unstructured pool until a command is given to generate a design. One improvement is that all different orderings of inputs yield identical results. Other improvements are that outputs are easily provable from inputs, and errors and inconsistencies in the input are more visible in the resulting data structure.

Although less rigorous than the second implementation, the first method is probably better for human designers. The second method requires storage of an unrelated set of facts (assertions), something which computers do well but people don't.

4 EXTENSIONS.

This chapter describes further work that can be done along the lines set out in the preceding chapters. I will first discuss immediate extensions to the existing systems for retrieval program generation and data structure design. There are also two sub-sections that describe extension of the techniques into other data base programming areas: data base update and data base restructuring.

4.1. Extensions to the DMLP.

The extensions discussed below are immediate because a conceptualization of their implementation in the existing systems is quite clear. I would estimate an average implementation time of 1 week per itemized extension, or approximately 3 man-months for all those itemized below.

4.1.01 Generation of the Data Division.

The program generator creates only a Procedure Division. There are several other divisions needed to make a complete COBOL program. The most significant of these is the Data Division. The reason for non-generation of the Data Division is that the major portion is generated by the Sub-Schema processor defined by the DBTG. Incorporation of the major portion of the Data Division can therefore be accomplished by including an appropriate COPY statement. A Data Division so derived must be extended only to include temporary working storage locations that are used by the Procedure Division.

The generated Data Division would appear as follows:

```
DATA DIVISION.
FILE SECTION.
FD DUMMY COPY <sub-schema-library-file>.
WORKING-STORAGE SECTION.
77 X1 PIC <> USAGE <>.
    .
    .
    .
77 Xn PIC <> USAGE <>.
```

The <sub-schema-library-file> must be supplied by the user. The X1 through Xn are the working storage variables used by the Procedure Division. Picture and Usage clause parameters will be the equivalent of the Picture and Usage clauses for the source data name if the variable is used to calculate a total, average, minimum or maximum of the item identified by the source data name. Otherwise the variable will have "PICTURE 9(10) USAGE COMP".

4.1.02 Disjunctive port selection.

In the present system, if a calculated key is used in a disjunction such that it is used as a port, then it must be so used in every disjunct. This requirement can be relaxed by requiring only that every disjunct must contain a port. Note that this requirement does not state that each such port must refer to the same calculated key.

4.1.03 Recursive processing.

A data base that contains a structure of the type given in Figure 4-1 can contain record types that are recursively linked to themselves. For example, the Bill of Materials parts structure for a manufacturing or assembly operation can be stored in this type of data base structure

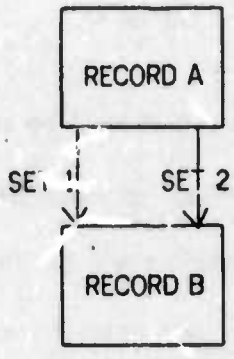


Figure 4-1. Recursive network structure.

In its present state, the system is "almost" capable of processing such a structure to a predetermined number of levels but in a very inelegant way; the user must specify each level to be processed, and one or more separate program paragraphs will result for each such level.

A much more elegant generated program would contain recursively entered paragraphs, such that each level can be processed by the same program segment. This type of program generation would require special recognition of

a recursive structure, incorporation of stack logic, and inclusion in HI-IQ of special commands for this type of retrieval.

4.1.04 Allow set name specification by the user.

The system should allow knowledgeable users to specify set names, e.g. define portions of the access path. This would allow resolution of possible ambiguities by the user, and user override of the program generator where appropriate.

This improvement would be necessary to erase the "almost" in the second paragraph of 4.1.03, above.

4.1.05 Arithmetic and temporary items.

This would allow the user to specify the calculation of values from data base items. Temporary items would be implemented to store the results of arithmetic and also to allow calculations or tests involving several occurances of the same item.

4.1.06 Distinguish ERRORSTATUS values.

At present, the system examines ERRORSTATUS only for a non-zero condition, and then assumes that an expected error condition, such as end of set, has occurred. By examining the actual value of ERRORSTATUS, the system can determine if the condition is as expected or truly represents an error that should be reported to the user.

A preferred implementation would require modifications to the DBTG specification. This preferred implementation would provide for an error filter resident in the data management routines but controlled by the run unit. This filter could be set to ignore certain conditions, trap on other conditions, and report a third set of conditions directly to the user.

As an alternative, such an error filter could also be directly incorporated as a standard section of the generated Procedure Division.

4.1.07 Efficient use of sorted sets.

DBTG provides for sets whose members are ordered on key values. Sequential searches that are dependent on the key value can be terminated when the key value of the current set member is higher than the key value range for the search condition.

4.1.08 Ordering of conjuncts and disjuncts.

When a compound conjunction is to be tested, the resulting program will be more efficient if the conjuncts are ordered so that the cheaper and most restrictive tests are performed first. Determining the best order is a complex task since the cheapest test may not be the most restrictive.

Disjuncts should be ordered with the cheapest and least restrictive test first.

A preferred implementation would require modifications to the DBTG specification. This preferred implementation would provide for an error filter resident in the data management routines but controlled by the run unit. This filter could be set to ignore certain conditions, trap on other conditions, and report a third set of conditions directly to the user.

As an alternative, such an error filter could also be directly incorporated as a standard section of the generated Procedure Division.

4.1.07 Efficient use of sorted sets.

DBTG provides for sets whose members are ordered on key values. Sequential searches that are dependent on the key value can be terminated when the key value of the current set member is higher than the key value range for the search condition.

4.1.08 Ordering of conjuncts and disjuncts.

When a compound conjunction is to be tested, the resulting program will be more efficient if the conjuncts are ordered so that the cheaper and most restrictive tests are performed first. Determining the best order is a complex task since the cheapest test may not be the most restrictive.

Disjuncts should be ordered with the cheapest and least restrictive test first.

Possible measures of test cost are the number of lines of generated code, the number of performs in the generated code, or the number of data access statements required for the test.

A determination of expected restrictiveness is more complicated although a certain simple-minded ordering is possible with equality tests rated as most restrictive, then weak inequalities, and finally strong inequalities. Further determination of restrictiveness would be more costly and would require knowledge of data base content. See the next paragraph.

4.1.09 Ordering of test arguments.

Tests are always of the form (A REL B), where "A" and "B" have some determinable value and REL is an equality or one of the inequality relations. If only one of A or B requires a statistic calculation in order to determine its value, then it is more efficient to determine the value of the other (non statistical) argument first. This is so because, if the value of the non statistical argument is known, then it is frequently possible to determine the truth value of the test without calculating the exact value of the statistic. See section 2.9.

4.1.10 Stronger system tests.

It is possible to strengthen the test used to terminate the calculation of a statistic (as discussed above) if the type of REL is taken into consideration. See section 2.9.

4.1. Extensions to the Retrieval Program Generator.

4.1.11 Use of data characteristics.

This extension is discussed in detail in section 2.9.

4.1.12 Removal of equivalent paragraphs.

As can be seen in Figure 2-31, where paragraphs 105 and 301 are equivalent, the system may generate duplicate paragraphs.

A "quicky" solution would be to search the resulting program for duplicate paragraphs. It would be more elegant to catalog each paragraph as a subroutine as it is generated. Each such catalogued paragraph then becomes a theorem that can be used for subsequent program generation.

4.1.13 Combination of statistical calculations.

If several different statistics are to be calculated over the same domain then the generated program will contain separate program segments for the calculation of each type of statistic. That is, all totals over the same domain will be calculated within the scope of a single PERFORM. However, if a count is also to be calculated over the same domain it will be counted in a separate section of the program that is invoked by a different perform.

This method of program generation leads to larger programs than necessary and can have an extremely negative effect on run-time efficiency.

A solution would require the detection of similar calculation domains by the Request Handler. This can be done by checking for similar LINKS assertions prior to making a new LINKS assertion. It would be necessary to change the LINKS assertions so that multiple actions can be specified.

4.1.14 Interfaces to externally coded program segments.

As it stands now the system generates a complete Procedure Division. The data retrieved by the generated program is simply displayed. In an actual operating environment, it may sometimes be desirable to process the retrieved data with a report writer, perform complex analyses, or output it to other devices. This suggests using the program generator to write sections of procedure that combined with other sections form a complete Procedure Division.

The machine generated sections would serve as data locators. Rather than displaying the data, such sections would simply assure its presence in Working Storage.

4.2 Extensions to the Data Structure Designer.

4.2.1 Generate complete design.

The designer ignores many data characteristics that might result in the design of a more efficient data base. A data base is more efficient if it requires less storage space, or if the total time required to process information therein is reduced.

There are also design issues, unrelated to the structural design, that must be considered for a complete data base design. Extending the designer to consider these issues and to make use of data characteristics is a proper next step.

4.2.2 Practicality of paradigm.

The "practicality" of accumulating a complete set of queries prior to attempting the design is open to question and should be investigated.

4.2.3 Further cost study.

Empirical study of the costs of human performance on equivalent design tasks is needed to determine if automatic data base design is less costly.

4.2.4 Recursive structures.

There is a particular substructure in network data bases for the storage of recursively linked records (Figure 4-1). If the programmer is to be extended as in section 4.1.03, then the designer should be extended in an equivalent fashion.

4.2.5 Explore alternative designs.

The data structure design arrived at by the system may be one of many possible designs for the set of queries. An extension to allow the generation of alternative designs (or at least indicate to the DBA which portions of the design are fixed and which are not) is suggested.

4.3 Automation of Data Base Update.

The traditional view of update is that it consists of three types of tasks; record addition, record deletion, and record modification. The latter two tasks are essentially similar to retrieval because a retrieval of the record in question must precede the execution of the actual update. Not much modification of the programmer would be needed to provide it with an update capability for deletion and modification.

Since records in a network data base may be linked to many other records, addition of records requires that all of the links be properly established. A record addition must therefore be preceded by several record retrievals, each retrieval locating one of the records to which the new record is to be linked. Code generation for these retrievals has already been demonstrated. Determination of the set of records to be retrieved and the criteria for their retrieval is still an unsolved problem.

This problem does not appear to be too difficult, however. The set of records is determined by finding all HIERARCHYGROUPs that contain the record to be added. The criteria for the retrieval of this set of records cannot be determined without some knowledge of the LOCATION MODE parameters defined in the Schema. The system would therefore have to be extended so that it can "understand" these parameters.

An update program generator should also be influenced in its programming by the batch vs on-line dichotomy. The batch processing type of program cannot communicate with the user in realtime, a restriction that does not apply to the on-line program. The system already has a limited capability for interactive information retrieval, and it should be possible to handle on-line transaction processing in the same way.

Batch updating requires that the structure of both the transaction and the data base be defined. Although I have not investigated this in detail, I am confident that the methods used for retrieval program generation are applicable without major extension.

4.4 Data Base Restructuring.

Restructuring becomes necessary when errors or oversights are detected in the current data structure or when changes in demand require a change in data structure. Such a data structure change may make certain programs obsolete and may also require that the data itself be manipulated and restructured. Restructuring can therefore be very costly, especially for large data bases.

The DMLP can have a beneficial impact on the costs of restructuring by reducing the reprogramming effort, thereby reducing programming costs and elapsed time. In addition to this immediate benefit, the technology and theory developed in the preceding chapters suggest a method that can totally eliminate the disruption usually associated with restructuring. The disruption is eliminated by "dissolving" the restructuring process in the normal usage of the data base.

An explanation of this method requires the introduction of a new concept; data structure generations. Whenever a structure is modified, the old structure description is not discarded; instead the new structure is recognized as a new structure generation and is assigned an incremental generation number. The current structure generation number is used to mark records when they are stored so that the appropriate structure description can be used when these records are later retrieved. Under this scheme multiple generations of structure would be allowed to exist simultaneously in the data base.

Programs are also marked with the identifying number of the generation in effect when the programs were written.

The system is therefore capable of detecting differences between the assumed structure in a program and the actual structure in the data base by comparing structure generation markers. As a result, whenever a program attempts to navigate through some structure that belongs to a different generation than the program, the system provides a subprogram that makes the data structure differences transparent to the program. The previous chapters have shown how these subprograms can be generated whenever the generation differences are limited to attribute migration or hierarchical inversion.

Generation data structures may also differ in the paths that exist between record types. Again, this is a difference easily resolved.

Differences that are not so easily resolvable are structures that differ in possible information content. In such cases the system must ask for human intervention or operate from a set of well-known assumptions. There may be other structural changes that cannot be handled automatically.

Implementation of this type of system provides a choice regarding the actual reformatting of the data. In highly volatile data bases actual reformatting can be indefinitely postponed because of the rapid decay of out-of-date structures.

A second alternative is to cause automatic reformatting of out-of-date structures as they are encountered in the course of normal processing.

The latter method assumes that "local" restructuring is feasible. I would expect situations to occur in which this assumption is not valid. Since all data is linked together, "chain reactions" might occur: a single change causes access to another record that must also be modified, causing access to another out-of-date structure, etc.

5 COST EFFECTIVENESS OF AUTOMATIC PROGRAMMING.

The true costs of this application of Artificial Intelligence cannot be measured until the application process is completed, that is, until the program generator is used with an actual data base to solve actual retrieval problems. Some measurement of program generation and data structure design has been done however. Considering the fact that the implementation utilizes interpreted LISP and Micro-Planner, both of which are notoriously expensive [Sussman and McDermott 1972]. these costs appear to be quite promising.

5.1 Measures of Automatic Program Generation.

Table 5-1 gives values of various measures of the program generation process. The queries and their corresponding programs are named P1 through P6. The first column of the table contains a measure of the size of the problem statement. The values in this column indicate the number of input lines containing entries other than "NIL" (a terminator) entered for the query. The next two columns are a measure of the complexity of the programming task; indicated for each program is the number of rules the system attempted to use and the number of such attempts that were successful.

	PROB LINES	RULES USE	RULES SUC	COBOL LINES	COBOL PARAS	RUN TIME	K-CORE SEC	$
P1	10	58	49	-	-	-	-	-
P2	13	84	77	32	8	4:09.1	5988	64.70
P3	14	118	102	67	11	6:22.5	9229	97.87
P4	16	128	107	61	11	5:26.2	7862	82.56
P5	20	99	88	30	4	3:14.4	4681	49.19
P6	12	79	62	29	4	2:55.9	4230	44.34
TOTAL		566	485	219	38	22:08.1	31990	338.66

Table 5-1. Costs of automatic program generation.

Columns five and six measure the size of the resulting COBOL programs in terms of both lines of code and number of paragraphs produced. The lines produced count is conservative: lines which label paragraphs or sections are not included in the count. Furthermore, lines of excessive length which are continued on a subsequent line are only counted as a single line.

One observation here is that the size of the problem statement is not a good measure of the size of the resulting program nor of the generation cost. A large problem statement may result in a relatively small program (as is the case with P5) if the relations in the problem statement correlate highly with records in the data base.

The final three columns of Table 5-1 indicate some direct measures of cost of the DMLP. The first of these indicates run time in minutes and seconds, the second column indicates the number of kilo-core-seconds used; and the last

column gives a total dollar charge for the usage as calculated on Carnegie-Mellon's DEC-10 computer. The primary component of the charge incurred by the program generator is the kilo-core-seconds used because it uses a lot of storage. The Carnegie-Mellon charge is approximately $.01 per kilo-core-second. This compares to an in-house charge of $.0007 per kilo-core-second at First National City Bank and an outside vendor charge of $.03 per kilo-core-second [Summers 1974], all for DEC-10's.

The cost measure has only been obtained for the generation of the last five programs in the table. These five programs contain 219 lines of COBOL source code (including DML statements) and cost a total of $338.66 in computer time to generate. This gives a mean cost of $1.55 per line of code with a range of $1.35 to $2.02 per line generated.

It is tempting to compare the cost of $1.55 per line of code with equivalent values measured for human programmers. Before I do so, some cautionary statements are in order. The figure of $1.55 per line is a variable cost and does not take into consideration the costs of development of the DMLP. Secondly, the cost to derive the problem statement, a human cost, is not included. Thirdly, it is certainly possible that the program obtained may turn out to be unsatisfactory because of errors or omissions in the problem statement. Such a condition will require a further iteration of program generation, and each such iteration will add an additional $1.55 per line to the

cost of the final program. Obviously, an extension to the system which would allow it to modify previously generated programs without a complete regeneration would be of great value.

5.2 Cost Factors in Programming.

The cost of the DMLP will be compared with costs calculated from four different sources. The predicted number of man months and computer usage for 1000 lines of COBOL code are given for each source in Table 5-2. A very detailed study of cost factors was carried out at the System Development Corporation and has been reported in [Nelson 1966; LaBolle 1966]. This study, also discussed in [Sharpe 1969], is based on observations of 12 COBOL programming projects. I have included both the minimum and the mean cost factors from the SDC study.

	NELSON (mean)	BRANDON	DELANEY	BAKER MILLS	NELSON (min)
MAN MO.	4.83	7.0	6	1.72	.27
COMP HRS.	18.08	20.5	25	7.22	1.18

Table 5-2. COBOL cost factors for 1000 source lines.

The Brandon study [Brandon 1963] gives normative (standard) values for resource expenditures for programming in COBOL on the IBM 7080. Brandon

provides an algorithm for calculating these values which is dependent on the complexity as well as the size of the program. Brandon's algorithm has a fixed cost component. As a result, the cost per line will be somewhat lower for lengthier programs, all other things being equal. The figures given in Table 5-2 were linearly extrapolated from values obtained from the algorithm for a 280 line COBOL program of average complexity.

Delaney's algorithm [Delaney 1966] is simply based on a productivity rate of 10 lines of code per man day, plus a 25% additional cost for program checkout. Computer time is caculated at the rate of 1 hour per man week. Delaney does not indicate how his figures were obtained. The productivity rate quoted by Delaney is applicable to programming in machine code. The SDC study found that programmers generate COBOL code at a slower rate than machine code, so Delaney's figures may be somewhat optimistic for COBOL programming.

The fourth study [Baker and Mills 1973] reports the results obtained from a single programming project at IBM. The computer hours used in this project were not indicated, so the figure given in the table has been estimated using the one-hour-per-man-week rule of thumb. Baker and Mills claim that the improvement in productivity is due to special management (Chief Programmer Teams) and programming (structured) techniques. Since the earlier SDC study showed a very large performance variance, we cannot attach much significance to the single observation provided by Baker and Mills.

All four of these sources considered only the programmer's task in calculating the cost factors. Other costs, such as problem definition cost and design iteration costs, were not included. Although the Brandon and Delaney algorithms have a documentation cost component, the values in Table 5-2 do not include this cost since the DMLP does not generate documentation.

Table 5-2 is fairly consistent with the DMLP as regards the basis for the calculations. There are, however, some costs that will be incurred with the DMLP that are not accounted for in Table 5-2. These costs consist of an investment cost associated with the DMLP, and the cost to phrase the information retrieval problem in terms of HI-IQ. These costs, especially the latter, are not negligible, but empirical study is needed to determine actual values.

5.3 Comparison of Dollar Costs.

By assigning some dollar costs to the factors in Table 5-2 it becomes possible to directly compare the various cost predictions (experiences) with the costs observed for the DMLP. Such a comparison is given in Table 5-3. The minimum cost line is based on a man-month cost of $1000. The reasonable cost line is based on $3000 per man-month (derived from an annual salary of $18,000 plus 100% for burden and overhead).

	DMLP	NELSON (mean)	BRANDON	DELANEY	BAKER MILLS	NELSON (min)
MIN COSTS	108	6,638	9,050	8,500	2,422	1,450
REAS COSTS	1,550	23,530	31,250	30,500	8,770	6,710

Table 5-3. $ Cost for 1000 lines of COBOL in 1974.

Both lines in Table 5-3 also take into consideration the machine cost factor. It is more difficull to determine appropriate charges for computer usage, because the computer hour usage figures in Table 5-2 were obtained several years ago. On the one hand, computer hours to implement 1000 lines of COBOL might be less today because of the improved efficiency of computer hardware. Conversely, debugging time might have increased because of the increased complexity of the programming environment. Furthermore, because of multi-programming, computer costs are no longer a simple function of CPU time.

As a minimum value, I will use a charge of $100 per 1963-1966 equivalent computer hour. This assumes an efficiency improvement factor of 1/5. As a reasonable value, I will use a charge of $500 per computer hour. This assumes that the factors discussed above in combination with inflation have cancelled each other out. Table 5-3 compares the minimum and reasonable dollar costs based on the cost factors of Table 5-2 with the costs experienced with the DMLP.

The reasonable cost for the DMLP is based on the charge of $.01 per k-core second actually experienced (the cost also includes minor charges for connect time, etc.). The minimum cost figure is based on the charge of $.0007 per k-core second used for the in-house DEC-10 at First National City Bank.

The DMLP has a dollar cost saving of 92% to 98% in the minimum cost frame. In the reasonable cost frame, the DMLP reduces cost by 77% to 95%.

This reduction in cost leaves a lot of room for absorption of the costs associated with the DMLP. Note that the costs for the DMLP as actually experienced are equivalent to the minimum charges applied to the most efficient project observed by Nelson; i.e., a "super" programmer working for $12,000 a year, incurring no burden or overhead.

5.4 Execution Costs.

The costs discussed so far are only program generation costs and ignore execution costs. The execution of computer generated code may be more or less efficient than the execution of programs written by people. We would like to think that there is some person who can write a better program than a machine for any problem. But, the DMLP need only be better than the average programmer for a net advantage to result from using it.

5.5 Cost of Data Structure Design.

It is difficult to judge the cost savings from automatic structure design since first, the automatic designer only partially replaces the human designer. Second, this task is quite new and cost experience is not yet available. The cost of automatic design seems quite low, however. The costs given in table 5-4 were obtained from the DEC-10 computer at Carnegie-Mellon University and are based on the same charging algorithm as described above.

The design process occurs in two steps. Simple fact accumulation from the set of queries is done first. Then the actual design is done from the accumulated facts. The fact extraction process consists of reading the query and extracting from it the implicit relationships it contains. Table 5-4 shows that 41 facts were extracted from a set of six queries containing 55 statements at a cost of $15.93. The costs given here were obtained from the example given In Chapter Three.

The second stage of the design process determines a network structure that will accomodate all of the necessary relations as captured in the facts base. This process is more complex than fact accumulation and requires almost twice as much time and a cost of $28.71.

INPUT	QTY	RUN TIME	$
STMNTS	55	1:15.65	15.93
FACTS	41	2:15.27	28.71
TOT		3:30.92	44.64

Table 5-4. Cost of Generating a Design.

The savings potential of these systems is high enough to justify this and further research.

A revision of the rules used by the DMLP so that they can be used by people is suggested. This would provide a method of structured programming for data base programmers. Baker and Mills attributed much of their success to structured programming.

6 ON THE APPLICATION OF ARTIFICIAL INTELLIGENCE.

Most successful Artificial Intelligence (AI) programs excercise their problem solving ability in unrealistic environments or apply it to a game playing task. Many of these applications are also not objective since the problem environment has been selected and constrained by the AI researcher. Actual replacement of human problem solvers with the machine is usually of no value to anyone except the AI community. Critics of AI can frequently stand unrefuted because much of what they claim is true: practical and objective applications of AI are few and far between.

To find in this criticism an indictment of AI is unrealistic and unfair. As many of the proponents of AI have claimed in defense, it is necessary for AI to cut its teeth on toy problems, games, and in otherwise limited environments before it can proceed to fill its highly touted promises. Perhaps the only fair criticism is that this teeth cutting is taking longer than envisioned by early workers in the field.

It is my feeling that AI is entering an age where application becomes feasible. By my statements here I do not mean to imply that there have not been valuable spin-offs from AI that have contributed to advances in fields such as computer science, linguistics, and psychology. However, there have not been many direct applications of complete AI technologies (with perhaps the exception of the pattern recognition field) to problem areas existing outside the

domain of AI, and especially outside the academic domain.

My own experience is perhaps a good demonstration of the coming of age of AI. The background I bring to this research is not primarily in AI, but lies more in the area of commercial data processing, data base management, and information retrieval. My experience in these areas predates by several years my introductory exposure to AI. I think this distinction is important because the problems which will be described shortly, existed for me long before I discovered the technology to solve them. As such, my circumstances provided a degree of objectivity not available to the AI researcher who is looking for problems to adapt to his technology.

6.1 Data Management Application.

This thesis describes a system which generates information retrieval programs and data structure declarations. These two problem solving tasks have been around for a long time, but growing commercial usage of network data bases has suddenly increased their complexity. (When I refer to a network data base I do not necessarily mean a data base on a computer network, but one in which a data item may be linked to many other data items in rings, hierarchies or other structures.) There has been an attempt within the United States and Canada to develop a standard for network data bases through CODASYL which spawned the inter-academia-industry Data Base Task Group (DBTG). This group has produced a specification for a general purpose network data base system

[CODASYL 1971a]. Although not yet accepted as standard, it is believed by many that the DBTG specification will be so adopted, perhaps by default since many computer manufacturers and software firms are now selling implementations of the DBTG specification.

Both applications described herein function in the data base environment as specified by the DBTG. Data structures designed by the system conform to DBTG specifications, and the generated retrieval programs are COBOL Procedure Divisions containing Data Manipulation Language (DML) as defined by the DBTG.

This adherence to external specifications is another important characteristic that distinguishes this application from other less practical and less objective applications.

6.1.1 Programming in network data bases.

Introduction of the network type of data base brings with it a new level of complexity for the business programmer. As Early [Early 1972] points out, instead of the usual logical-physical dichotomy which a programmer traditionally deals with, there is a logical(a)-logical(b)-physical trichotomy facing the network data base programmer. The logical(a) level corresponds to data relationships as perceived by the data user. The logical(b) level concerns the actual links between the data in the data base.

Bachman recognizes this problem in the 1973 ACM Turing Award Lecture [Bachman 1973] where he likens the programmer's problem (in the network data base environment) to that of a navigator. The programmer is given the user's desired data and relations; his task is to find the proper access path through the myriad of connective links in the data base.

This problem, the programmer's problem, should be reduced. After all, much of the problem is caused by an artifact introduced by the data network, the data links. It should be noted here that others in the field, primarily Codd [Codd 1970, 1972a, 1972b], have proposed relational models of data bases whereby they hope to avoid the emergence of this artifact. It is not clear to me however, how this can be done without exactly the kind of problem solving program which is the subject of this paper.

Finding a path to certain pieces of data while satisfying certain constraints is not at all dissimilar to a complicated robot/maze task. This similarity may be a partial explanation for the direct applicability of AI technology to the problem.

The technology used is not robot technology however. Instead, the solution was found in the automatic programming area. Specifically, I used a compiler developed by Buchanan and Luckham [Buchanan and Luckham 1974; Buchanan 1974] to create the data base retrieval program generator. The retrieval program generator, consisting primarily of a set of Micro-Planner theorems

[Hewitt 1971; Sussman and Winograd 1972], is compiled by the APG system from a set of rules. These rules are stated in a formalism corresponding to Hoare's logic of programs [Hoare 1969]. Each rule describes a loop, an assignment statement, a Data Manipulation Language Statement, or other program construct. A rule describes the effect of a program construct and the conditions under which it may be used. For example, the rule for the GET statement states that (a), it is not possible to GET a record unless its location in the data base is known to the data management system and (b), following the GET, the record is located in working storage. Other rules, especially ones dealing with the assemblage of larger program blocks, do not bear such close resemblance to sections of a programming manual as does the rule for GET. These other rules capture my own programming knowledge which I had previously acquired and developed as a professional programmer. It was a difficult exercise to express my knowledge in the APG formalism.

The rules are translated to Micro-Planner theorems. The resulting program (consisting of Micro-Planner theorems) is an information retrieval program generator. One of its inputs is the above mentioned data base structure definition, the other input is a retrieval query. The output is a COBOL Procedure Division.

The program generator is data base independent, and may generate different programs for identical queries if the structure is different. This is why the data

base structure is one of the inputs to the program generation process. The design of this structure is the subject of the next section.

6.1.2 Designing network data structures.

Another problem attendant to network data base management is data base structure design. That is, to design appropriate links to exist between data elements so that data relationships can be properly captured and reconstructed. Since all such relationships must be implicitly contained in the set of all queries, it should be possible to derive a data base structure from such a set.

The automatic data structure designer operates in exactly this fashion. It generates a data structure design that is satisfactory for a set of queries.

The power of the design program lies in two areas. First, it has the ability to recognize hierarchies and inversions thereof. Detection of a hierarchy that is processed "both ways" leads to the inclusion of a confluent hierarchy in the designed structure. For example, a confluent hierarchy would be used to store information regarding doctors and patients since doctors can have several patients, and conversely, patients can have several doctors.

The designer also determines the location of data items (attributes) in the hierarchical structure. For example, hospital name can be a patient attribute (where he is being treated), a doctor attribute (where he works), and of course

a hospital attribute. If hospital-patient and hospital-doctor hierarchies (several patients and several doctors per hospital) have already been constructed by the system, then it will assign the attribute in question to the hospital record. This permits usage of hospital name as a unique attribute of hospital, doctor, and patient.

In Early's words, the system has found a logical(b) arrangement which accomodates all logical(a) relationships. As a matter of fact, translation from logical(a) to logical(b) also characterizes the system's activities as a programmer. Further translation to the physical level is accomplished by the data management system, e.g. an implementation of the DBTG specification. As was the case with the program generator, the system is defined with a set of rules that are translated by a compiler to a program of Micro-Planner theorems. However, in this case the resulting system is not a code generator, it is a declarative generator.

6.2 Acquisition and Representation of Knowledge.

If a system is to apply knowledge, it must first obtain such knowledge. The ongoing work in "learning" systems is partially motivated by the desire to minimize the effort of building the knowledge base for a system which must understand a particular domain. Until generalized learning systems become available which can be adapted in a practical way for use with a particular application domain, it is necessary to imbed knowledge of the application either

by building a specific learner for that system or by providing the system directly with the knowledge it needs. The trouble with the former approach is that building the learner may turn out to be a more complex task than building the rest of the system.

The latter approach is the one that was taken for the applications discussed here. The rule formalism provides a very good framework for capturing network data base programming knowledge. Even so, capturing the knowledge was not a straightforward process and involved many trial-and-error iterations.

The occurrence of so much trial-and-error even when using this formalism is perhaps surprising. The problem arises because much of the existing knowledge (residing in the application specialist's head) does not have a well defined structure. It is possible to map the specialist's knowledge to a structured set of formal rules in several ways, all of which are valid, but some of which lead to an extremely inefficient program generation process.

However, some of the knowledge does have a well defined structure, for example the knowledge regarding GET as discussed above. In general, the rules for single program statements are very similar to usage rules that might be found in a reference manual. It is the rules that control the next higher level of programming, the construction of program blocks, that are complex and not easily derived.

Since the content of the rules determines the decision tree that controls the search space, the choice of mappings mentioned above can have a dramatic impact on the efficiency of the search for a program. If the use of a particular rule occurs far out in the branches of the tree, and if the rule is a powerful discriminator, then it would be appropriate to consider an alternative rule structure that would permit earlier evaluation of the rule in question.

Notice that this makes it very important that the applications specialist be very closely integrated with the AI technologist. Close integration will allow the specialist's knowledge to be captured in a way that is most effective for the AI methodology being used. This task is complicated if the AI representation shows little correspondence to the "natural" representation.

In the particular application discussed in this paper, the integration was accomplished by training the application specialist (myself) to become an AI technologist. I can only imagine the problems that would result, especially regarding efficiency, if there is less integration and the two functions are embodied in different individuals.

Table 5-1 illustrates the impact of close integration on the efficiency of the resulting application. For example, of the 84 rules that were tried for the generation of program P2, only 7 were unnecessary. For all six programs, out of 566 rules tried, 485 were successful. In other words, on average only 14%

of the search takes place in fruitless branches of the tree.

At the present stage of AI technology, successful applications will require a high degree of integration of the AI technologist and the applications specialist. Until psychologists discover how people store knowledge, it will be necessary for the applications specialist to "come to the machine" and describe his processes in AI terminology. Conversely, the AI technologist will need to acquire a great deal of knowledge about the application.

7 THE RELATIONAL AND NETWORK MODELS OF DATA BASES.

7.1 Introduction.

Several recent papers on relational and network data base models discuss these models in a choice framework. That is, these papers make the assumption that a choice of one model is to be made, and then argue the supremacy of one model over the other. Codd and Date [Codd and Date 1974; Date and Codd 1974] recently argued for the relational model, others [Jardine 1974] have argued the other way.

I find it unnecessary to argue this point. Obviously, the relational model provides a much better applications interface and a better framework for the discussion of data bases. But I ask the rhetorical question: How do proponents of the relational model propose to implement general relational data bases without utilizing some type of network structure? Contemplation of this question reveals my central premise which is best illustrated by analogy - the relational model is to the network model as COBOL is to assembly language.

Viewing the two models in this fashion leads to the realization that there is a need for both models and suggests an approach whereby the relational model can be developed and incorporated by users in an orderly fashion.

7.2 Levels of Data Structure Description.

Codd [Codd 1970] originally introduced the relational model as a means for

higher level data structure description. However, just as COBOL and other high level languages must be translated to a machine level sequence of instructions, so must relational representations be translated to access path descriptions. Early [Early 1972] was the first to formalize the levels of description attendant to data bases. Early defined three levels of description:

(a) relational
(b) access path
(c) implementation

The two highest levels comprise, in Early's terms, the semantic level of description.

Although the access path and implementation levels are usually invisible to the user of a relational level model, the user should demand standardization at all three levels, and especially at the access path level. If he does not get such standardization, he will find that his data base is not transportable across machines or across various implementations of the relational model.

There are many possible choices of models at the access path level, including sequential file, indexed file and network data bases. The most general of these is the network model, and the DBTG report represents a standardization of this model.

The network model, primarily as embodied in the DBTG report and its predecessor, IDS [Bachman and Williams 1964], has enjoyed far greater

acceptance in implementation than has the relational model. This success is not due to a flaw in the relational model, but is simply caused by the limitations of technology. Since the relational model captures a higher level of generality, it requires further advancements in technology for implementation. In terms of the preceding analogy, assemblers preceded compilers for essentially similar reasons.

7.3 Translation Approach.

Because of the increased acceptance and use of DBTG systems, and the need for using a standard access path model, the next logical step in the development of the relational model is to devise translators for the translation of relational level descriptions to DBTG access path descriptions. This allows development of the relational model on the base of existing DBTG implementations.

I call this the translation approach and differentiate it from the total approach. The total approach is the approach taken with implementations of the relational model to date. That is, each such implementation is self-contained, and access path level and implementation level descriptions are hidden from the user. Such implementations of the relational model therefore usually differ not only at the user (relational) interface, but also in the storage structure (access paths) of the data.

7.4 Advantages of translation.

The translation approach offers several advantages over the total approach:

(a) protects current users of DBTG
(b) is extensible with minimized disruption
(c) permits user flexibility
(d) permits orderly development of the relational model

Standardization at the access path level, as was urged above, is facilitated with the translation approach. These advantages all acrue because the translation approach is well defined (or locked-in) at the access path level.

Once a user's relational description has been translated to a DBTG access path description (e.g. COBOL - DML program), this description then exists within a fixed framework. Subsequent changes in the relational technology or standards will not affect the user's operations. If he chooses to, the user can re-translate selected relational descriptions to take advantage of such modifications.

If the necessary relational technology has not yet been developed for a certain application, the user can always fall back on the access path model and manually generate the necessary access path descriptions, or he can modify previously translated access path descriptions so that his application is properly processed. The same options are available for the improvement of inefficiencies caused by the generality of the relational model.

7.5 Two Implemented Translators.

The DMLP and the automatic designer are translators. The HI-IQ language (which both accept as input) allows users to specify queries without worrying about the structure of the data base. HI-IQ is a relational language because it lets the user specify queries in terms of the relationships he perceives.

Codd might not agree to this classification of HI-IQ because it is not based on his relational calculus or algebra and because it does not permit the naming of relations. This means that multiple direct relationships between two entities cannot be specified because a name is needed to distinguish between them.

Such multiple direct relationships are infrequent in commercial processing. In any case, extension of HI-IQ and the two systems to permit the naming of relations would not be difficult.

BIBLIOGRAPHY.

Bachman, C. W. (1965), "Software for Random Access Processing", Datamation, April 1965, pp36-41.

Bachman, C. W. (1969), "Data Structure Diagrams", Data Base 1, 2, Quarterly newsletter of ACM-SIGBDP.

Bachman, C. W. (1973), "The Programmer as Navigator", CACM 16, 11, November 1973, pp653-658.

Bachman, C. W. and S. B. Williams (1964), "A General Purpose Programming System for Random Access Memories", Proceedings, Fall Joint Computer Conference 1964, pp411-422.

Baker, Terry F. and Harlan D. Mills (1973), "Chief Programmer Teams", Datamation, December 1973, pp58-61.

Boyce, R. F., D. D. Chamberlin, W. F. King III , and M. M. Hammer (1973), "Specifying Queries as Relational Expressions: Square", IBM Research RJ 1291, October 1973.

Brandon, Dick H., (1963), Management Standards for Data Processing, Van Norstrand Co., Princeton, N.J., 1963.

Buchanan, J. R. (1974), "A Study in Automatic Programming", PhD Thesis, Stanford University.

Buchanan, J. R. and D. C. Luckham (1974), "On Automating the Construction of Programs", Stanford AI Project Memo, Stanford University.

Childs, D. L. (1968), "Feasibility of a set-theoretical data structure - a general structure based on a reconstituted definition of a relation", Proc. IFIP congress, 1968, North Holland Publishing Co., Amsterdam, pp162-172.

CODASYL (1970), CODASYL COBOL Journal of Development, 1969, available from ACM, New York City; IFIP Administration Data Processing Group, Amsterdam; and British Computer Society, London.

CODASYL. (1971a), CODASYL Data Base Task Group April 1971 Report, available from ACM, New York City; IFIP Administration Data Processing Group, Amsterdam; and British Computer Society, London.

CODASYL (1971b), Feature Analysis of Generalized Data Base Management Systems, April 1971, available from ACM, New York City; IFIP Administration Data Processing Group, Amsterdam; and British Computer Society, London.

BIBLIOGRAPHY.

Codd, E. F. (1970), "A Relational Model of Data for Large Shared Data Banks", CACM 13, 6, June 1970, pp377-387.

Codd, E. F. (1972a), "Further Normalization of the Data Base Relational Model, in R. Rustin (Ed), Data Base Systems, Prentice Hall, 1972.

Codd, E. F. (1972b), "Relational Completeness of Data Base Sublanguages", in R. Rustin (Ed), Data Base Systems, Prentice Hall, 1972.

Codd, E. F. and C. J. Date (1974), "Interactive Support for Non-programmers: The Relational and Network Approaches", IBM Research RJ-1400, June 6, 1974.

Dahl, O. J., E. W. Dijkstra, and C. A. R. Hoare (1972), Structured Programming, Academic Press, New York.

Date, C. J. and E. F. Codd (1974), "The Relational and Network Approaches: Comparison of the Application Programming Interfaces", IBM Research RJ-1401, June 6, 1974.

Delaney, William A. (1966), "Predicting the Costs of Computer Programs", Data Processing Magazine, October 1966, pp32-34.

Early, J. (1972), "On the Semantics of Data Structures", in R. Rustin (Ed), Data Base Systems, Prentice Hall, 1972.

General Electric (1965), Introduction to Integrated Data Store, Publication CPB-1048, General Electric Company Information Systems Department, Phoenix, Arizona.

Gerritsen, R. (1974), "Automatically Generated Programs for Information Retrieval; IRP, a Rudimentary System", Carnegie-Mellon University, Graduate School of Industrial Administration, W.P.-47-73-74.

Gibson, T. A. and P. F. Stockhausen (1973), "MASTER LINKS-Hierarchical Data System", The Bell System Technical Journal 52, 10, December 1973, pp1691-1724.

Hewitt, C. (1971), "Description and Theoretical Analysis of Planner", PhD Thesis, Massachusetts Institute of Technology.

Hoare, C. A. R. (1969), "An Axiomatic Basis for Computer Programming", CACM 12, 10, October 1969, pp576-580.

Igarashi, S, R L. London and D. C. Luckham (1973), "Automatic Program Verification I: A Logical Basis and Implementation", Stanford A.I. Memorandum 200, May 1973.

Informatics (1969), MARK IV File Management System Reference Manual, Informatics Inc., Document No. SP-69-810-1.

Jardine, Donald A. (Ed) (1974), Data Base Management Systems, Proceedings of the Share Working Conference on Data Base Management Systems, Montreal, Canada, July 23-27, 1973, North-Holland Publishing Company, 1974, pp157-160.

La Bolle, V. (1966), "Development of Equations for Estimating the Costs of Compuler Program Production", System Development Corporation Technical Memorandum TM-2918/000/00, April 5, 1966.

Lavalee, P. A., S. Ohayon and R. Sauvain (undated), "DMS Data Base Strategies for Interrogation and Update", Xerox, Technology Report.

McCarthy, J., P. W. Abrahams, D. J. Edwards, T. P. Hart and M. I. Levin (1972), LISP 1.5 Programmer's Manual, M.I.T. Press.

McKeeman, W., J. J. Horning, and D. B. Wortman (1970), A Compiler Generator, Prentice Hall, New Jersey.

Naur, P., et al. (1960), "Report on the Algorithmic Language ALGOL 60", CACM 3, p299.

Nelson, E. A. (1966), "Management Handbook for the Estimation of Computer Programming Costs", System Development Corporation Technical Memorandum TM-3225/000/00, October 31, 1966.

Newell, A., J. C. Shaw, and H. A. Simon (1960), "Report on a General Problem-Solving Program for a Computer", Information Processing, proceedings, International Conference of Information Processing, Paris: UNESCO, 1960, pp256-264.

Postley, John A. (1969), "General Purpose Systems: The MARK IV File Management System", in Fred Gruenberger (Ed), Critical Factors in Data Management, Prentice Hall, New Jersey, 1969.

Puerling, B. W. and J. T. Roberts (1973), "The Natural Dialogue System". The Bell Systems Technical Journal 52, 10, December 1973, pp1725-1741.

Sharpe, William F. (1969), The Economics of Computers, Columbia University Press, New York, 1969.

Shefter, H. (1958), Faster Reading Self-Taught, Washington Square Press, New York.

Simon, H. A. (1961), "Experiments with a Heuristic Compiler", the RAND Corp., Report P-2349.

Simon, H. A. and Laurent Siklossy (1972), Representation and Meaning; Experiments with Data Processing Systems, Prentice Hall, Englewood Cliffs, N.J.

Summers, Allen L. (1974), "The Captive Computer Utility", Computer Decisions 6, 1, January 1974.

Sussman, Gerald J. and Drew V. McDermott (1972), "Why Conniving is Better than Planning", Massachusetts Institute of Technology, Artificial Intelligence Memo 255A, April 1972.

Sussman, Gerald J. and Terry Winograd (1972), "Micro-planner Reference Manual", M.I.T. Project MAC Report.

Teichroew, Daniel (1970), "Problem Statement Languages in MIS", Proceedings, International Symposium of BIFOA, "Management Information Systems - A Challenge to Scientific Research", Cologne, July 1970, pp253-270.

Teichroew, Daniel and Hasan Sayani (1971), "Automation of System Building", Datamation, August 15, 1971, pp25-30.

Waldinger, R. J. and R. C. T. Lee (1969), "PROW: A Step Toward Automatic Program Writing", presented at the International Joint Conference on Artificial Intelligence, 1969.

Xerox (1970), XDS Sigma Data Management System (DMS) Reference Manual, Document 90-17-38 A, Xerox Data Systems, El Segundo, California.